XL Fortran for AIX

ランゲージ・リファレンス

バージョン 8.1.1

SC88-9395-01
(英文原典：SC09-4947-01)

XL Fortran for AIX

ランゲージ・リファレンス

バージョン 8.1.1

SC88-9395-01
(英文原典：SC09-4947-01)

ご注意!

本書および本書で紹介する製品をご使用になる前に、特記事項に記載されている情報をお読みください。

本書は、IBM XL Fortran Version 8.1 for AIX (プログラム番号 5765-F70) の修正レベル 1 および新版において特に断りのない限り、それ以降のすべてのリリースに適用されます。製品のレベルに合った版であることを確かめてご使用ください。

本マニュアルに関するご意見やご感想は、次の URL からお送りください。今後の参考にさせていただきます。

http://www.ibm.com/jp/manuals/main/mail.html

なお、日本 IBM 発行のマニュアルはインターネット経由でもご購入いただけます。詳しくは

http://www.ibm.com/jp/manuals/ の「ご注文について」をご覧ください。

(URL は、変更になる場合があります)

お客様の環境によっては、資料中の円記号がバックスラッシュと表示されたり、バックスラッシュが円記号と表示されたりする場合があります。

原 典: SC09–4947–01
XL Fortran for AIX
Language Reference
Version 8.1.1

発 行: 日本アイ・ビー・エム株式会社

担 当: ナショナル・ランゲージ・サポート

第1刷 2003.6

この文書では、平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、平成角ゴシック体™W5、および平成角ゴシック体™W7を使用しています。この(書体*)は、(財) 日本規格協会と使用契約を締結し使用しているものです。フォントとして無断複製することは禁止されています。

注* 平成明朝体™W3、平成明朝体™W9、平成角ゴシック体™W3、
平成角ゴシック体™W5、平成角ゴシック体™W7

目次

XL Fortran for AIX xiii
OpenMP Fortran API、バージョン 2.0 の機能 xiii
Fortran 95 の機能 xiii
XL Fortran の変更個所の要約 xiv
強調表示の規則 xv

本書の使い方 xvii
強調表示の規則 xvii
構文図の読み方. xix
構文図. xix
構文図の例 xx
本書の例についての注意事項 xx
関連資料 xxi
標準仕様 xxi

第 1 部 XL Fortran 言語 1

第 1 章 IBM XL Fortran for AIX の概要 . . 3
XL Fortran (XLF). 3
OpenMP とは何か 4
コンパイラー・ディレクティブ 5
ライブラリー・ルーチン 5
環境変数 5
Fortran 95 とは何か 5
FORALL 5
PURE 6
ELEMENTAL 6
初期化 6
仕様関数 6
削除された機能 7
Fortran 90 とは何か 7
Fortran 90 自由ソース形式 7
パラメーター化されたデータ型 7
派生型 8
配列の機能強化 8
ポインター. 8
動的処理 8
制御構造体の機能強化 8
プロシージャーの機能強化 9
モジュール. 9
新規の組み込みプロシージャー. 10
有効な XL Fortran プログラムと無効な XL Fortran プログラム. 10

第 2 章 言語エレメント. 11
文字. 11
名前. 12
ステートメント 13
ステートメント・キーワード 14
ステートメント・ラベル 14
行およびソース形式 14
固定ソース形式 16
自由ソース形式 19
IBM 自由ソース形式 21
条件付きコンパイル 22
ステートメントおよび実行の順序 24

第 3 章 データ型およびデータ・オブジェクト 27
データ型 27
型付きパラメーターおよび指定子 27
データ・オブジェクト. 28
定数. 28
自動オブジェクト 29
組み込み型 29
整数. 29
実数. 31
複素数 33
論理. 36
文字. 37
バイト 41
派生型 41
入出力 47
派生型のタイプの決め方 47
レコード構造. 55
UNION および MAP 59
タイプなしリテラル定数 63
16 進定数. 63
8 進定数 64
2 進定数 65
ホレリス定数. 65
タイプなし定数の使用方法 66

タイプの決め方 69
変数の定義状況 70
　定義を発生させるイベント 70
　未定義を発生させるイベント 74
割り振り状況. 77
変数のストレージ・クラス 78
　基本ストレージ・クラス 78
　2 次ストレージ・クラス 79
　ストレージ・クラスの割り当て. 79

第 4 章 配列の概念 83
配列. 83
　次元の境界 84
　次元のエクステント 84
　配列のランク、形状、およびサイズ . . . 85
配列宣言子 85
明示的形状配列 86
　明示的形状配列の例 87
　自動割り付け配列 87
　整合配列 88
　ポインティング先配列. 88
想定形状配列. 89
　想定形状配列の例 89
据え置き形状配列 90
　割り振り可能配列 90
　配列ポインター 91
想定サイズ配列 92
　想定サイズ配列の例 93
配列エレメント 94
　注 94
　配列エレメントの順序. 95
配列セクション 95
　添え字トリプレット 97
　ベクトル添え字 99
　配列セクションおよびサブストリングの範囲 100
　配列セクションおよび構造体コンポーネント 100
　配列セクションのランクおよび形状 . . . 102
配列コンストラクター 102
　配列コンストラクターの暗黙 DO リスト 103
配列にかかわる式. 105

第 5 章 式および割り当て 107
はじめに. 107
　1 次子 108
定数式 109
　定数式の例 109
初期化式. 110
　初期化式の例 110
宣言式 111
　宣言式の例 112
演算子および式 113
　一般式 113
　算術式 113
　文字 116
　関係式 117
　論理 119
　1 次子 122
拡張組み込みおよび定義済み演算 122
式の計算. 123
　演算子の優先順位. 123
　BYTE データ・オブジェクトの使用法 . . 126
組み込み割り当て. 127
　算術変換. 129
WHERE 構造体 131
　マスクされた配列割り当ての解釈 . . . 133
FORALL 構造体 139
　FORALL 構造体の解釈 140
ポインターの割り当て 142
　ポインター割り当ての例 144
　整数ポインターの割り当て. 144

第 6 章 制御構造 147
ステートメント・ブロック. 147
IF 構造体 147
　例 149
CASE 構造体 149
　例 152
DO 構造体 152
　終端ステートメント 153
DO WHILE 構造体 157
　例 157
分岐 157

第 7 章 プログラム単位およびプロシージャー 159
有効範囲. 159
　名前の有効範囲 160
関連付け. 164
　ホスト関連付け 164
　使用関連付け 166

ポインター関連付け 167
整数ポインター関連付け 168
プログラム単位、プロシージャー、およびサブプログラム 169
内部プロシージャー 169
インターフェースの概念 170
インターフェース・ブロック 172
インターフェースの例 175
総称インターフェース・ブロック 176
明白な総称プロシージャー参照 176
総称インターフェース・ブロックによる組み込みプロシージャーの拡張 178
定義済み演算子 179
定義済み割り当て 180
メインプログラム 181
モジュール 183
モジュールの例 186
ブロック・データのプログラム単位 187
ブロック・データ・プログラム単位の例 188
関数およびサブルーチン・サブプログラム 188
プロシージャー参照 190
組み込みプロシージャー 191
組み込みプロシージャー名と他の名前の不一致 192
引き数 193
実引き数の仕様 193
引き数関連付け 196
%VAL および %REF 197
仮引き数の意図 199
オプションの仮引き数 200
指定されていないオプションの仮引き数に対する制限事項 200
文字引き数の長さ 201
仮引き数としての変数 201
仮引き数として割り振り可能なオブジェクト 203
仮引き数としてのポインター 204
仮引き数としてのプロシージャー 205
仮引き数としてのアスタリスク 206
プロシージャー参照の解決 206
名前に対するプロシージャー参照の解決の規則 207
総称名に対するプロシージャー参照の解決 208
再帰 209
純粋プロシージャー 209
例 211
エレメント型プロシージャー 212
例 214

第 8 章 I/O の概念 217
レコード 217
定様式レコード 217
不定様式レコード 218
ファイル終了レコード 218
ファイル 218
外部ファイル 218
外部ファイル・アクセス・モード: 順次、直接、およびストリーム 219
内部ファイル 221
装置 222
装置の接続 222
データ転送ステートメントの実行 . . . 223
データ転送ステートメントの非同期の実行 225
アドバンス I/O および非アドバンス I/O 227
データ転送が行われる前後のファイルの位置 227
条件および IOSTAT 値 229
レコードの終わり条件 230
ファイルの終わり条件 230
エラー条件 230

第 9 章 I/O の形式設定 239
形式指示の形式設定 239
データ編集記述子 239
制御編集記述子 241
文字ストリング編集記述子 242
編集 243
複素数編集 244
データ編集記述子 244
A (文字) 編集 244
B (2 進) 編集 245
E、D、および Q (拡張精度) 編集 . . . 247
EN 編集 249
ES 編集 250
F (指数なし実数) 編集 251
G (一般) 編集 253
I (整数) 編集 255
L (論理) 編集 256
O (8 進数) 編集 257
Q (文字カウント) 編集 259
Z (16 進) 編集. 260
制御編集記述子 262

/ (スラッシュ) 編集 262
: (コロン) 編集 262
$ (ドル記号) 編集. 263
アポストロフィ / 二重引用符編集 (文字ストリング編集記述子) 263
BN (ブランク・ヌル) および BZ (ブランク・ゼロ) 編集. 264
H 編集 265
P (スケール因数) 編集 266
S、SP、および SS (符号制御) 編集 . . . 267
T、TL、TR、および X (定位置) 編集 . . 268
I/O リストと形式仕様の相互作用. 269
リスト指示の形式設定 270
リスト指示入力 271
リスト指示出力 272
名前リストの形式設定 274
名前リスト入力データ 274
名前リスト出力データ 280

第 10 章 ステートメントおよび属性 . . . 285
属性 289
ALLOCATABLE 289
ALLOCATE 290
ASSIGN 293
AUTOMATIC 294
BACKSPACE 296
BLOCK DATA 298
BYTE 299
CALL 302
CASE 304
CHARACTER 306
CLOSE 312
COMMON 314
COMPLEX 319
CONTAINS 324
CONTINUE 325
CYCLE 325
DATA. 327
DEALLOCATE 331
派生型 (TYPE) 333
DIMENSION 334
DO. 335
DO WHILE 337
DOUBLE COMPLEX 339
DOUBLE PRECISION 342
ELSE 346
ELSE IF 347
ELSEWHERE 348
END 350
END (構造体) 351
END INTERFACE. 354
END TYPE 356
ENDFILE. 357
ENTRY 359
EQUIVALENCE 362
EXIT 365
EXTERNAL 366
FORALL 367
FORALL (構造体). 371
FORMAT. 372
FUNCTION 378
GO TO (割り当て) 382
GO TO (計算) 384
GO TO (無条件) 385
IF (算術). 386
IF (ブロック) 387
IF (論理). 388
IMPLICIT 389
INQUIRE. 392
INTEGER 399
INTENT 405
INTERFACE. 407
INTRINSIC 409
LOGICAL 411
MODULE 416
MODULE PROCEDURE. 417
NAMELIST 418
NULLIFY 419
OPEN 420
OPTIONAL 427
PARAMETER 429
PAUSE 430
POINTER (Fortran 90) 431
POINTER (整数) 434
PRINT 436
PRIVATE 438
PROGRAM 441
PROTECTED 442
PUBLIC 444
READ. 445
REAL 453
RECORD. 458

RETURN 460
REWIND 461
SAVE 463
SELECT CASE 465
SEQUENCE 467
ステートメント関数 468
STATIC 470
STOP 472
SUBROUTINE 473
TARGET 475
TYPE 476
タイプ宣言 481
USE 488
| VALUE 491
VIRTUAL 493
VOLATILE 493
WAIT 496
WHERE 497
WRITE 500

第 11 章 汎用ディレクティブ 507
注釈形式および非注釈形式ディレクティブ 507
注釈形式ディレクティブ 507
非注釈形式ディレクティブ 510
ディレクティブおよび最適化 510
断定ディレクティブ 511
ループ・アンロール用ディレクティブ . . 511
ディレクティブの詳細説明 511
ASSERT 511
CNCALL 513
COLLAPSE 515
EJECT 516
INCLUDE 517
INDEPENDENT 519
#LINE 523
PERMUTATION 525
@PROCESS 526
SNAPSHOT 527
SOURCEFORM 529
| STREAM_UNROLL 530
SUBSCRIPTORDER 532
| UNROLL 534
| UNROLL_AND_FUSE 536

第 12 章 組み込みプロシージャー 539
組み込みプロシージャーのクラス 539
照会組み込み関数 539
エレメント型組み込みプロシージャー . . 539
システム照会組み込み関数 541
変換組み込み関数 541
組み込みサブルーチン 541
データ表示モデル 542
整数ビット・モデル 542
整数データ・モデル 543
実データ・モデル 544
組み込みプロシージャーの詳しい記述 . . . 545
ABORT () 545
ABS (A) 546
ACHAR (I) 547
ACOS (X) 547
ACOSD (X) 548
ADJUSTL (STRING) 549
ADJUSTR (STRING) 549
AIMAG (Z), IMAG (Z) 550
AINT (A, KIND) 551
ALL (MASK, DIM) 552
ALLOCATED(ARRAY) または
ALLOCATED(SCALAR) 553
ANINT (A, KIND) 553
ANY (MASK, DIM) 554
ASIN (X) 555
ASIND (X) 556
ASSOCIATED (POINTER, TARGET) . . . 557
ATAN (X) 558
ATAND (X) 559
ATAN2 (Y, X) 559
ATAN2D (Y, X) 561
BIT_SIZE (I) 562
BTEST (I, POS) 563
CEILING(A, KIND) 564
CHAR (I, KIND) 565
CMPLX (X, Y, KIND) 566
CONJG (Z) 567
COS (X) 568
COSD (X) 569
COSH (X) 569
COUNT (MASK, DIM) 570
CPU_TIME(TIME) 571
CSHIFT (ARRAY, SHIFT, DIM) 573
CVMGx (TSOURCE, FSOURCE, MASK) . . 574
DATE_AND_TIME (DATE, TIME, ZONE,
VALUES) 576

DBLE (A) 578
DCMPLX (X, Y) 579
DIGITS (X) 580
DIM (X, Y) 581
DOT_PRODUCT (VECTOR_A, VECTOR_B) 582
DPROD (X, Y). 582
EOSHIFT (ARRAY, SHIFT, BOUNDARY,
DIM) 583
EPSILON (X) 585
ERF (X) 586
ERFC (X) 587
EXP (X) 588
EXPONENT (X) 589
FLOOR(A, KIND) 589
FRACTION (X) 591
GAMMA(X). 592
GETENV (NAME, VALUE) 593
HFIX (A) 594
HUGE (X) 595
IACHAR (C) 595
IAND (I, J) 596
IBCLR (I, POS) 597
IBITS (I, POS, LEN) 598
IBSET (I, POS). 599
ICHAR (C) 600
IEOR (I, J) 600
ILEN (I) 601
IMAG (Z) 602
INDEX (STRING, SUBSTRING, BACK) . . 602
INT (A, KIND). 603
INT2(A) 605
IOR (I, J) 606
ISHFT (I, SHIFT) 607
ISHFTC (I, SHIFT, SIZE) 608
KIND (X) 609
LBOUND (ARRAY, DIM) 609
LEADZ (I) 610
LEN (STRING). 611
LEN_TRIM (STRING) 612
LGAMMA (X) 612
LGE (STRING_A, STRING_B). 613
LGT (STRING_A, STRING_B). 614
LLE (STRING_A, STRING_B). 615
LLT (STRING_A, STRING_B). 616
LOC (X) 617
LOG (X) 618
LOG10 (X) 619
LOGICAL (L, KIND). 619
LSHIFT (I, SHIFT) 620
MATMUL(MATRIX_A, MATRIX_B,
MINDIM) 621
MAX (A1, A2, A3, ...) 623
MAXEXPONENT (X). 625
MAXLOC(ARRAY, DIM, MASK) または
MAXLOC(ARRAY, MASK). 625
MAXVAL(ARRAY, DIM, MASK) または
MAXVAL(ARRAY, MASK). 627
MERGE (TSOURCE, FSOURCE, MASK) . . 629
MIN (A1, A2, A3, ...) 630
MINEXPONENT (X) 631
MINLOC(ARRAY, DIM, MASK) または
MINLOC(ARRAY, MASK) 632
MINVAL(ARRAY, DIM, MASK) または
MINVAL(ARRAY, MASK) 634
MOD (A, P). 635
MODULO (A, P) 636
MVBITS (FROM, FROMPOS, LEN, TO,
TOPOS) 637
NEAREST (X,S) 638
NINT (A, KIND) 639
NOT (I) 640
NULL(MOLD) 641
NUM_PARTHDS(). 642
NUMBER_OF_PROCESSORS (DIM) 643
NUM_USRTHDS(). 644
PACK (ARRAY, MASK, VECTOR) 644
PRECISION (X) 646
PRESENT (A) 647
PROCESSORS_SHAPE () 648
PRODUCT(ARRAY, DIM, MASK) または
PRODUCT(ARRAY, MASK) 648
QCMPLX (X, Y) 650
QEXT (A) 651
RADIX (X) 652
RAND (). 652
RANDOM_NUMBER (HARVEST) 653
RANDOM_SEED (SIZE, PUT, GET,
GENERATOR) 654
RANGE (X). 656
REAL (A, KIND) 657
REPEAT (STRING, NCOPIES). 658
RESHAPE (SOURCE, SHAPE, PAD, ORDER) 658

RRSPACING (X) 660
RSHIFT (I, SHIFT) 660
SCALE (X,I) 661
SCAN (STRING, SET, BACK). 662
SELECTED_INT_KIND (R) 663
SELECTED_REAL_KIND (P, R) 664
SET_EXPONENT (X,I) 665
SHAPE(SOURCE) 666
SIGN (A, B) 666
SIGNAL (I, PROC) 668
SIN (X) 669
SIND (X) 670
SINH (X) 671
SIZE (ARRAY, DIM). 671
SIZEOF(A) 672
SPACING (X) 674
SPREAD (SOURCE, DIM, NCOPIES) 675
SQRT (X) 676
SRAND (SEED) 677
SUM(ARRAY, DIM, MASK) または
SUM(ARRAY, MASK) 678
SYSTEM (CMD, RESULT) 680
SYSTEM_CLOCK(COUNT, COUNT_RATE,
COUNT_MAX) 681
TAN (X) 682
TAND (X) 683
TANH (X) 684
TINY (X) 685
TRANSFER (SOURCE, MOLD, SIZE) 685
TRANSPOSE (MATRIX). 687
TRIM (STRING) 687
UBOUND (ARRAY, DIM) 688
UNPACK (VECTOR, MASK, FIELD) 689
VERIFY (STRING, SET, BACK) 690

第 2 部 XL Fortran でのマルチスレッド・プログラミング 693

第 13 章 SMP ディレクティブ 695
SMP ディレクティブの概要 695
並列領域構造体 695
作業共用構造体 696
結合された並列作業共用構造体 696
同期構造体 696
その他の OpenMP ディレクティブ . . . 696
非 OpenMP SMP ディレクティブ 697
SMP ディレクティブの詳細な説明 697
ATOMIC 697
BARRIER 700
CRITICAL / END CRITICAL 701
DO / END DO. 703
DO SERIAL. 707
FLUSH 708
MASTER / END MASTER 711
ORDERED / END ORDERED 712
PARALLEL / END PARALLEL 715
PARALLEL DO / END PARALLEL DO 718
PARALLEL SECTIONS / END
PARALLEL SECTIONS 722
PARALLEL WORKSHARE / END
PARALLEL WORKSHARE 726
SCHEDULE 726
SECTIONS / END SECTIONS. 730
SINGLE / END SINGLE 734
THREADLOCAL 738
THREADPRIVATE 741
WORKSHARE 747
OpenMP ディレクティブ文節 750
ディレクティブ文節のグローバル規則 . . 750
COPYIN 752
COPYPRIVATE 753
DEFAULT 754
IF 756
FIRSTPRIVATE 757
LASTPRIVATE. 758
NUM_THREADS 760
ORDERED 761
PRIVATE 762
REDUCTION 764
SCHEDULE 767
SHARED. 769

第 14 章 OpenMP 実行環境ルーチンおよびロック・ルーチン 773
omp_destroy_lock 774
omp_destroy_nest_lock 775
omp_get_dynamic 775
omp_get_max_threads 775
omp_get_nested 776
omp_get_num_procs 776
omp_get_num_threads 777
omp_get_thread_num 777

omp_get_wtick 778
omp_get_wtime 779
omp_in_parallel 779
omp_init_lock 780
omp_init_nest_lock 781
omp_set_dynamic 781
omp_set_lock 782
omp_set_nested 783
omp_set_nest_lock 783
omp_set_num_threads 784
omp_test_lock 784
omp_test_nest_lock 785
omp_unset_lock 786
omp_unset_nest_lock 786

第 15 章 Pthreads ライブラリー・モジュール 787
Pthreads のデータ構造、関数、およびサブルーチン 788
Pthreads のデータ構造 788
スレッド属性オブジェクトに操作を実行する関数 788
スレッドに操作を実行する関数およびサブルーチン 788
mutex 属性オブジェクトに操作を実行する関数 789
mutex オブジェクトに操作を実行する関数 789
条件変数の属性オブジェクトに操作を実行する関数 789
条件変数オブジェクトに操作を実行する関数 789
スレッド固有データに操作を実行する関数 789
制御スレッド取り消し機能に操作を実行する関数およびサブルーチン 790
1 回限りの初期化の操作を実行する関数 790
f_maketime 790
f_pthread_attr_destroy 790
f_pthread_attr_getdetachstate 791
f_pthread_attr_getguardsize 791
f_pthread_attr_getinheritsched 792
f_pthread_attr_getschedparam. 792
f_pthread_attr_getschedpolicy. 793
f_pthread_attr_getscope 793
f_pthread_attr_getstackaddr 794
f_pthread_attr_getstacksize 795
f_pthread_attr_init 795
f_pthread_attr_setdetachstate 796
f_pthread_attr_setguardsize 796
f_pthread_attr_setinheritsched. 797
f_pthread_attr_setschedparam 798
f_pthread_attr_setschedpolicy 798
f_pthread_attr_setscope 799
f_pthread_attr_setstackaddr 800
f_pthread_attr_setstacksize 800
f_pthread_attr_t 801
f_pthread_cancel 801
f_pthread_cleanup_pop. 801
f_pthread_cleanup_push 802
f_pthread_cond_broadcast 803
f_pthread_cond_destroy 804
f_pthread_cond_init. 804
f_pthread_cond_signal 805
f_pthread_cond_t 805
f_pthread_cond_timedwait. 805
f_pthread_cond_wait 806
f_pthread_condattr_destroy 807
f_pthread_condattr_getpshared 807
f_pthread_condattr_init. 808
f_pthread_condattr_setpshared 808
f_pthread_condattr_t 809
f_pthread_create. 809
f_pthread_detach 810
f_pthread_equal 811
f_pthread_exit 811
f_pthread_getconcurrency 812
f_pthread_getschedparam 812
f_pthread_getspecific 813
f_pthread_join 813
f_pthread_key_create 814
f_pthread_key_delete 814
f_pthread_key_t 815
f_pthread_kill 815
f_pthread_mutex_destroy 816
f_pthread_mutex_getprioceiling 816
f_pthread_mutex_init 816
f_pthread_mutex_lock 817
f_pthread_mutex_setprioceiling 818
f_pthread_mutex_t 818
f_pthread_mutex_trylock 818
f_pthread_mutex_unlock 819
f_pthread_mutexattr_destroy 819
f_pthread_mutexattr_getprioceiling 820

f_pthread_mutexattr_getprotocol 820
f_pthread_mutexattr_getpshared 820
f_pthread_mutexattr_gettype 821
f_pthread_mutexattr_init 822
f_pthread_mutexattr_setprioceiling 822
f_pthread_mutexattr_setprotocol 823
f_pthread_mutexattr_setpshared 823
f_pthread_mutexattr_settype 824
f_pthread_mutexattr_t 825
f_pthread_once 825
f_pthread_once_t 826
f_pthread_rwlock_destroy 826
f_pthread_rwlock_init 826
f_pthread_rwlock_rdlock 827
f_pthread_rwlock_t 828
f_pthread_rwlock_tryrdlock 828
f_pthread_rwlock_trywrlock 829
f_pthread_rwlock_unlock 830
f_pthread_rwlock_wrlock 830
f_pthread_rwlockattr_destroy 831
f_pthread_rwlockattr_getpshared 831
f_pthread_rwlockattr_init 832
f_pthread_rwlockattr_setpshared 832
f_pthread_rwlockattr_t 833
f_pthread_self 833
f_pthread_setcancelstate 833
f_pthread_setcanceltype 834
f_pthread_setconcurrency 834
f_pthread_setschedparam 835
| f_pthread_setspecific 836
f_pthread_t 836
f_pthread_testcancel 837
f_sched_param 837
f_sched_yield 837
f_timespec 837

第 3 部 XL Fortran 言語ユーティリティー 839

第 16 章 浮動小数点制御および照会のプロシージャー 841
fpgets fpsets 841
浮動小数点制御および照会のための効果的なプロシージャー 842
xlf_fp_util 浮動小数点プロシージャー . . 845
IEEE モジュールとサポート 849
コンパイルと例外処理 850
IEEE をインプリメントするための一般規則 851
IEEE 派生データ型と定数 851
IEEE 演算子 853
IEEE プロシージャー 854
浮動小数点状況に関する規則 874
例 875

第 17 章 サービス・プロシージャーおよびユーティリティー・プロシージャー . . . 879
一般的なサービス・プロシージャーおよびユーティリティー・プロシージャー 879
サービス・プロシージャーおよびユーティリティー・プロシージャーのリスト 881
| alarm_ 881
| bic_ 882
| bis_ 882
| bit_ 883
clock_ 883
| ctime_ 883
date 884
dtime_ 884
etime_ 885
exit_ 885
| fdate_ 885
fiosetup_ 886
flush_ 887
| ftell_ ftell64_ 887
getarg 888
| getcwd_ 888
| getfd 889
getgid_ 889
getlog_ 890
getpid_ 890
getuid_ 890
global_timef 891
gmtime_ 892
| hostnm_ 892
iargc 893
idate_ 893
ierrno_ 893
irand 894
irtc 894
itime_ 895
jdate 895

lenchr_ 895
lnblnk_ 896
ltime_ 896
mclock 897
qsort_ 897
qsort_down 898
qsort_up 898
rtc 899
setrteopts 899
sleep_ 900
time_ 900
timef 900
timef_delta 901
umask_ 901
usleep_ 902
xl_ _trbk 902

第 4 部 ハードウェアと XL Fortran 903

第 18 章 ハードウェア・ディレクティブと組み込みプロシージャー 905
ハードウェア固有のディレクティブ 905
CACHE_ZERO 905
ISYNC 906
LIGHT_SYNC 906
PREFETCH 906
ハードウェア固有の組み込みプロシージャー 910
FCFI(I) 910
FCTID(X) 910
FCTIDZ(X) 911
FCTIW(X) 911
FCTIWZ(X) 912
FMADD(A, X, Y) 912
FMSUB(A, X, Y) 913
FNABS(X) 913
FNMADD(A, X, Y) 914
FNMSUB(A, X, Y) 914
FRES(X) 915
FRSQRTE(X) 915
FSEL(X,Y,Z) 916
MTFSF(MASK, R). 916
MTFSFI(BF, I) 917
MULHY(RA, RB) 917
ROTATELI(RS, IS, SHIFT, MASK) . . . 917
ROTATELM(RS, SHIFT, MASK) 918
SETFSB0(BT) 918
SETFSB1(BT) 918
SFTI(M, Y) 919
TRAP(A, B, TO) 919

第 5 部 付録 921

付録 A. 異なる標準の間の互換性 923
Fortran 90 の互換性 924
使用されなくなった機能 925
削除された機能 927

付録 B. ASCII 文字セットと EBCDIC 文字セット 929

特記事項 939
商標 941

用語集 943

指標 955

XL Fortran for AIX

この節では、IBM XL Fortran for AIX コンパイラーの強調すべき点について説明します。 XLF コンパイラーのその他のバージョン、または他の Fortran 95、Fortran 90、FORTRAN 77 コンパイラーに精通しているユーザーは、この節を参考にしてください。

OpenMP Fortran API、バージョン 2.0 の機能

OpenMP Fortran API では、従来の FORTRAN 77、Fortran 90 および Fortran 95 言語の標準仕様を補足するために使用することのできる追加機能が提供されています。詳細については、 4 ページの『OpenMP とは何か』を参照してください。

OpenMP アーキテクチャー検討委員会 (ARB) は、API の各局面に関する解釈についての質問に回答します。これらの質問の中には、このバージョンの XL Fortran コンパイラーで実現されているインターフェースの機能に関連するものもあります。インターフェースに関連した質問に対するこの委員会からの回答によっては、 XL Fortran コンパイラーの将来のリリースに変更が生じる場合があります。その変更によって、以前のリリースとの互換性が保てなくなる場合もあります。

Fortran 95 の機能

F95 Fortran 95 言語の標準は、削除された機能を除いて、 FORTRAN 77 と Fortran 90 言語の両方の標準に対する上位互換性があります。詳細については、 5 ページの『Fortran 95 とは何か』を参照してください。

Fortran 95 では、広範な言語機能も追加されています。これらの機能の詳細については、 5 ページの『Fortran 95 とは何か』を参照してください。 Fortran 標準化委員会は、Fortran の各局面の解釈についての質問に回答します。これらの質問の中には、XL Fortran コンパイラーで実現されている言語の機能に関連するものもあります。これらの言語の機能に関連した質問に対する上記の委員会からの回答によっては、XL Fortran コンパイラーの将来のリリースに変更が生じる場合があります。その変更によって、以前のリリースとの互換性が保てなくなる場合もあります。 F95

XL Fortran の変更個所の要約

この節では、XL Fortran (XLF) バージョン 8.1.1 と バージョン 8.1 の相違点について説明します。 XLF コンパイラーの以前のバージョン、または他の Fortran 95、Fortran 90、FORTRAN 77 コンパイラーに精通しているユーザーは、このセクションを参考にしてください。

XL Fortran バージョン 8.1.1 では、以下の新機能および変更機能を提供しています。

新しいコンパイラー・オプションおよびサブオプションは以下のとおりです。

- **-qport=sce** コンパイラー・オプションにより、選択された論理式でショート・サーキット評価を行うことができます。
- **-qprefetch** コンパイラー・オプションにより、プリフェッチ命令の自動挿入を制御できます。
- **-qpic** コンパイラー・オプションにより、位置独立コード生成の目次サイズを選択できます。
- **-qextname=name** サブオプションを使用することにより、グローバル・エンティティーの名前に下線を追加して明確に識別できます。
- **-qsuppress=cmpmsg** サブオプションが使用できるようになったため、コンパイルの進行と正常完了を報告するコンパイラー・メッセージをフィルターに掛けて除外できます。
- **intrinthds** ランタイム・オプションは **MATMUL** および **RANDOM_NUMBER** 組み込みプロシージャーの並列実行のスレッド数を指定します。

以下の XL Fortran の機能強化によりドラフト Fortran 2000 標準に適応しました。

- **OPEN**、**READ**、**WRITE**、および **INQUIRE** ステートメントが外部ストリーム・ファイルの記憶単位にアクセスするためのストリーム入出力メソッドをサポートするようになりました。
- **PROTECTED** 属性およびステートメントは、エンティティーと同一のモジュール内に定義されているモジュール・プロシージャーによってのみそのエンティティーを変更できることを保証します。
- **VALUE** 属性およびステートメントで仮引き数と実引き数の間の引き数関連付けを指定することにより、仮引き数に実引き数の値を引き渡すことができます。

以下のパフォーマンス関連ディレクティブが新たに追加されました。

- **STREAM_UNROLL** ディレクティブはコンパイラーに対して、ソフトウェア・プリフェッチとループ・アンロールを結合した機能を反復カウントの大きな **DO** ループに適用することを指示します。
- **UNROLL_AND_FUSE** ディレクティブは内側のループ本体を複製し、レプリカをアンロール・ループと結合します。これにより繰り返し回数が減り、自己の一時データの再使用によってデータの局所性が向上します。

- **UNROLL** ディレクティブを外部ループのアンロールに適用できるようになりました。

HTML および PDF バージョンのユーザーズ・ガイドおよびランゲージ・リファレンスでは、資料間の重要な相互参照がリンクされました。他方の資料に記載されている相互参照トピックを確認するために、一方の資料のトピックの確認を中断する必要はなくなりました。

本書の前版からの変更点は、左余白の縦線 (|) によって示されています。

強調表示の規則

本書では、旧版 (XLF バージョン 8.1) とは異なる強調表示の規則が用いられています。 Fortran 95 および XL Fortran 拡張のために以前使用されていた強調表示のための色は、以降のセクションで示されている *F95* および *IBM* アイコンに変更されました。詳細については、 xvii ページの『強調表示の規則』を参照してください。

本書の使い方

本書は、主に参照用として書かれており、Fortran またはプログラミングの学習を目的としたものではありません。本書は、Fortran の概念について知識を有し、以前に Fortran アプリケーション・プログラムを作成した経験をお持ちのプログラマーを対象としています。

本書は主に以下の 3 部から構成されています。

923 ページの『付録 A. 異なる標準の間の互換性』 では、FORTRAN 77、Fortran 90 および Fortran 95 標準間の上位互換性と下位互換性について説明されています。 929 ページの『付録 B. ASCII 文字セットと EBCDIC 文字セット』 では、ASCII および EBCDIC 文字セットを示しています。 943 ページの『用語集』 には、この文書で繰り返し使用されている用語の意味がアルファベット順に示されています。

本書で IBM 拡張と Fortran 95 情報を識別する方法の詳細については、『強調表示の規則』を参照してください。

強調表示の規則

本書では、以下の強調表示の規則を使っています。

- Fortran 90 の情報について、ここでは FORTRAN 77 資料と区別はしません。
- Fortran 90 に追加された Fortran 95 情報は、以下の方法のいずれかで表示されます。

  関係のある文章が小さいブロックである場合は、 F95 (FORTRAN 95 の開始) および F95 (FORTRAN 95 の終了) アイコンで囲まれます。たとえば、次のようになります。

  – F95 以下は Fortran 95 情報です。 F95

  関係のある Fortran 95 情報が大きなブロックである場合は、以下のように示されます。

  Fortran 95

  以下の段落には、Fortran 95 関連情報が含まれています。

  Fortran 95 の終り

- Fortran 90 または Fortran 95 標準に対する **IBM 拡張**は、以下のいずれかの方法で示されます。

関係のある文章が小さいブロックである場合は、▶ IBM (IBM 拡張の開始) および IBM ◀ (IBM 拡張の終了) のアイコンで囲まれています。たとえば、次のようになります。

– ▶ IBM 以下は、Fortran 90 または Fortran 95 に対する IBM 拡張です。

IBM ◀

Fortran 90 または Fortran 95 に対する **IBM 拡張**が大きなブロックである場合は、以下のように示されます。

IBM 拡張

以下の段落には、Fortran 90 または Fortran 95 標準に対する **IBM 拡張**が含まれています。

IBM 拡張 の終り

Fortran 90 または Fortran 95 標準に対する IBM 拡張には、以下のものが含まれます。

1. 標準仕様でプロセッサー依存値またはプロセッサー依存動作として説明されているものを XL Fortran にインプリメントしたもの
2. XL Fortran バージョン 8.1 に追加された情報で、Fortran 90 または Fortran 95 に含まれていないもの

Fortran のキーワード、コマンド、ステートメント、ディレクティブ、組み込みプロシージャー、コンパイラー・オプション、およびファイル名は太字で表されます。たとえば、**OPEN**、**COMMON**、および **END** などです。

他の情報ソースへの参照は、イタリックで示されています。変数名およびユーザー指定の名前は、たとえば *array_element_name* のように小文字のイタリックで示されます。

注: 本書で、新しい標準カテゴリーまたは拡張機能カテゴリーのいずれかに属する機能について説明するすべての部分が強調表示されているわけではありません。特定の機能について初めて説明している箇所だけが強調表示されます。たとえば、**AUTOMATIC** ステートメントに関する説明 (294 ページの『AUTOMATIC』を参照) は、強調表示されますが、本書で **AUTOMATIC** ステートメントを参照する他のすべての記述がそのように表示されるわけではありません。

特定の標準に属するエレメントを含む構文図については、その構文図に続くテキストで、またはその構文図自体の注で、適切に識別されています。たとえば、**ELSEWHERE** ステートメント (348 ページの『ELSEWHERE』) の構文図は、Fortran 90 標準に属します。ただし、このステートメントには、Fortran 95 オプション *where_construct_name* も含まれます。このオプションは、それに続くテキストと構文図の両方で表記規則に従って強調表示されます。

構文図の読み方

本書では、Fortran ステートメントとエレメントの構文は、通称「線路図」と呼ばれる表記方法により図解されています。

変数名またはユーザー指定の名前が *_list* で終わっている場合は、これらの項のリストをコンマで区切って指定できることを示しています。

句読記号、括弧、算術演算子、その他の特殊文字は、構文の一部として入力しなければなりません。

構文図

- 構文図は線の経路に沿って、左から右へ、上から下へ読みます。
  ►►── 記号は、ステートメントの始まりを示します。
  ──► 記号は、ステートメントが次の行へ続くことを示します。
  ►── 記号は、ステートメントが前の行から続いていることを示します。
  ──►◄ 記号は、ステートメントの終わりを示します。
  プログラム単位、プロシージャー、構造体、インターフェース・ブロック、および派生型の定義は、それぞれ数個の個別のステートメントから構成されています。そのような項目の場合、構文表現は 1 つのボックスで囲まれ、個々の構文図は、対応する Fortran ステートメントにとって必須の指定順序を示します。
- 必須項目は、次のように水平線 (主線) 上に示されます。

```
►►─keyword─required_argument───────────────────────────────────────────────►◄
```

- オプションの項目は、主線の下側に示されます。

```
►►─keyword──┬───────────────────┬──────────────────────────────────────────►◄
            └─optional_argument─┘
```

注: オプションの項目 (構文図中にないもの) には、大括弧の [と] が付けられます。たとえば、[UNIT=]u などのようになります。

- 複数の項目から選択が可能な場合、それらはスタック状に重ねて示されます。

  複数の項目から 1 つを*選択しなければならない* 場合は、スタック内の 1 つの項目が主線上に示されます。

```
►►─keyword──┬─required_argument─┬──────────────────────────────────────────►◄
            └─required_argument─┘
```

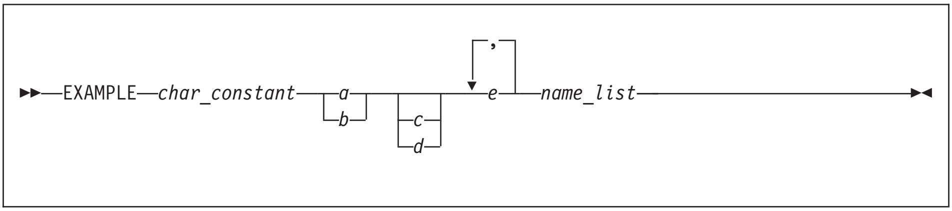
  項目の選択がオプションである場合は、スタック全体が主線の下側に示されます。

```
►►─keyword──┬───────────────────┬──────────────────────────────────────────►◄
            ├─optional_argument─┤
            └─optional_argument─┘
```

- 主線の左上に戻る矢印 (繰り返し矢印) は、繰り返し可能な項目を示します。その線上に示されるのはセパレーターです (ブランクの場合もある)。

スタック上の繰り返し矢印は、スタックから複数にわたり項目を選択できることを示します。

構文図の例



この構文図は次のように解釈します。

- キーワード **EXAMPLE** を入力します。
- *char_constant* に値を入力します。
- *a*、*b* いずれか一方の値のみを入力します。
- オプションとして、*c* または *d* のいずれかの値を入力します。
- *e* に少なくとも 1 つの値を入力します。複数の値を入力する場合は、それぞれの値の間にコンマが必要です。
- *name_list* に *name* の値を少なくとも 1 つ入力します。複数の値を入力する場合は、それぞれの値の間にコンマが必要です。 (*_list* 構文は、前の構文 (*e* の構文) と等しくなります。)

本書の例についての注意事項

- 本書で使用する例は、ストレージの節約、エラーのチェック、パフォーマンスの高速化などを意図しておらず、想定できるすべての使用法を示すことを目的とするものでもないので、簡単な形でコーディングされています。
- 本書の例は、呼び出しコマンド **f77**、**fort77**、**xlf**、**xlf_r**、 **xlf_r7**、**xlf90**、**xlf90_r**、**xlf90_r7**、**xlf95**、 **xlf95_r**、**xlf95_r7** のいずれかを使用してコンパイルされます。詳細については、「ユーザーズ・ガイド」の『*XL Fortran* プログラムのコンパイル』を参照してください。

- HTML セッションから編集セッションへサンプル・コードを貼り付けることができます。
- 本書のサンプル・プログラム、および本書に記載した例を表すその他のプログラムは、 **/usr/lpp/xlf/samples/modules** ディレクトリーの中に入っています。

関連資料

さらに詳しく知りたい場合は、以下の資料を参照してください。

- 「*XL Fortran for AIX ユーザーズ・ガイド*」では、XL Fortran を使用した Fortran プログラムのコンパイル方法、リンク方法、および実行方法を説明しています。
- 「*Engineering and Scientific Subroutine Library Guide and Reference*」では、科学技術計算サブルーチン・ライブラリー (ESSL) のルーチンについて説明しています。
- 「*AIX Technical Reference: Base Operating System and Extensions Volume 1*」と「*AIX Technical Reference: Base Operating System and Extensions Volume 2*」では、 AIX サブルーチン、システムの呼び出し、および BLAS (基礎線形代数サブルーチン) について説明しています。

標準仕様

XL Fortran は、以下の標準仕様に従って設計されています。本書で扱っているいくつかの機能について正確な定義が必要な場合は、これらの標準仕様を参照してください。

1. 「*American National Standard Programming Language FORTRAN*, ANSI X3.9-1978」。
2. 「*American National Standard Programming Language Fortran 90*, ANSI X3.198-1992」。 (本書では Fortran 90 という略式名で呼んでいます。この標準仕様は、下記 4 の ISO 標準仕様と同等です。)
3. 「*ANSI/IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985」。
4. 「*Information technology - Programming languages - Fortran*, ISO/IEC 1539-1:1991 (E)」。
5. 「*Information technology - Programming languages - Fortran - Part 1: Base language*, ISO/IEC 1539-1:1997」。 (本書では Fortran 95 という略式名で呼んでいます。)
6. 「*Information technology - Programming languages - Fortran - Floating-Point Exception Handling*, ISO/IEC JTC1/SC22/WG5 N1379」。
7. 「*Information technology - Programming languages - Fortran - Enhance Data Type Facilities*, ISO/IEC JTC1/SC22/WG5 N1378」。

8. 「*Military Standard Fortran DOD Supplement to ANSI X3.9-1978*, MIL-STD-1753 (United States of America, Department of Defense standard)」。 XL Fortran では、Fortran 90 の標準仕様に対して後から加えられた拡張機能のみをサポートしていることに注意してください。
9. 「*OpenMP Fortran Language Application Program Interface*, Version 2.0 (Nov 2000) specification」。

第 1 部 XL Fortran 言語

本節では、XL Fortran の主な概念、およびエレメントについて詳しく説明します。まず言語の概要として、バージョン 8.1.1 コンパイラー、およびサポートされている標準仕様について説明し、次に以下の言語概念について説明します。

- 言語エレメント
- データ型およびオブジェクト
- 配列
- 式および割り当て
- 制御構造
- プログラム単位およびプロシージャー
- I/O の概念
- I/O の形式設定

XL Fortran 言語の主要部分の説明に加え、この部には以下の章が含まれています。

- ステートメント
- 汎用ディレクティブ
- 組み込みプロシージャー

以下の部では、XL Fortran 言語に特有の特徴についてさらに説明しています。

- XL Fortran でのマルチスレッド・プログラミング
- XL Fortran 言語ユーティリティー
- ハードウェアと XL Fortran

第 1 章 IBM XL Fortran for AIX の概要

この章では、以下の項目について説明します。

- 『XL Fortran (XLF)』
- IBM 4 ページの『OpenMP とは何か』 IBM
- F95 5 ページの『Fortran 95 とは何か』 F95
- 7 ページの『Fortran 90 とは何か』
- 10 ページの『有効な XL Fortran プログラムと無効な XL Fortran プログラム』

本書は、XL Fortran コンパイラーのバージョン 8.1 の解説書です。 Fortran (FORmula TRANslation) は、主として数値計算のためのアプリケーション用に設計された高水準プログラム言語です。 Fortran は、科学、工学、および数学分野のほとんどのアプリケーションに適した言語です。

XL Fortran は、Fortran 95 の国際標準化機構 (ISO) 標準仕様、および同等の Fortran 95 の米国規格協会 (ANSI) 標準仕様をインプリメントしたものです。 XL Fortran は、その他の主要な業界標準も満たしています。 xxi ページの『標準仕様』を参照してください。

「ユーザーズ・ガイド」は、XL Fortran で作成されたプログラムのコンパイル方法、リンク方法、および実行方法を説明しています。他の情報については、 xxi ページの『関連資料』 を参照してください。

ツールについてのヒント

デバッグに **/usr/bin/idebug** コマンドを使用できます。このコマンドは、IBM® 分散デバッガーを起動します。

XL Fortran (XLF)

XLF 言語は、次のものから構成されています。

- Fortran 90

  Fortran 90 言語標準は、FORTRAN 77 言語標準をサポートしています。詳細については、 923 ページの『付録 A. 異なる標準の間の互換性』を参照してください。

Fortran 95

- Fortran 95

  Fortran 95 では、Fortran 90 と FORTRAN 77 言語の標準の機能のうちごく一部が削除されています。 FORTRAN 95 標準から削除された機能は、Fortran 標準化委員会によって、最新のプログラミングの業務にほとんど準拠していないと見なされたものです。ただし、削除された機能に代わる有効な機能があるので、機能そのものは Fortran 95 から削除されていません。詳細については、 927 ページの『削除された機能』を参照してください。 XL Fortran は、Fortran 95 標準仕様から削除された機能を引き続きサポートしています。その理由は FORTRAN 77、Fortran 90、および Fortran 95 との互換性を保つためです。

  また、Fortran 95 では、広範な言語機能も追加されています。これらの機能の一部を続く節で簡単に説明します。 Fortran 標準化委員会は、Fortran の各局面の解釈についての質問に回答します。これらの質問の中には、XL Fortran コンパイラーで実現されている言語の機能に関連するものもあります。これらの言語の機能に関連した質問に対する上記の委員会からの回答によっては、XL Fortran コンパイラーの将来のリリースに変更が生じる場合があります。その変更によって、以前のリリースとの互換性が保てなくなる場合もあります。

Fortran 95 の終り

IBM 拡張

- Fortran 言語に対する IBM および業界の拡張機能

  XL Fortran は、FORTRAN 77、Fortran 90、および Fortran 95 言語の標準仕様に対する多くの拡張機能をサポートしています。これらの拡張機能には、他のものに加えて、OpenMP 仕様のサポート、サービス・プロシージャーとユーティリティー・プロシージャーのサポート、および Pthreads ライブラリー・モジュールのサポートが含まれています。これら機能のいくつかは、次の節で簡単に説明されています。

IBM 拡張 の終り

OpenMP とは何か

IBM 拡張

OpenMP は、Fortran と C/C++ で移植可能な共用メモリーの並列処理を実現するための仕様です。 OpenMP 仕様では、並列環境を制御するために使用できる多くのコンパイラー・ディレクティブ、ライブラリー・ルーチン、および環境変数が提供されています。この節では、OpenMP の機能を簡単に概説します。

コンパイラー・ディレクティブ

XL Fortran には多くの汎用ディレクティブが含まれています。ハードウェア・プラットフォームに固有なディレクティブについては、 905 ページの『ハードウェア固有のディレクティブ』を参照してください。

OpenMP アプリケーション・プログラミング・インターフェースでは、並列領域、作業共用構造体、および同期構造体を定義する多くのディレクティブが提供されており、データを共用したり独占するためのサポートも提供されています。

詳細については、 695 ページの『第 13 章 SMP ディレクティブ』を参照してください。

ライブラリー・ルーチン

OpenMP API は、ランタイム実行環境を制御して照会するための一連のライブラリー・ルーチンや、一連の汎用ロック・ルーチン、およびポータブル・タイマー・ルーチンをサポートしています。

詳細については、 773 ページの『第 14 章 OpenMP 実行環境ルーチンおよびロック・ルーチン』を参照してください。

環境変数

OpenMP 環境変数では、ランタイム実行環境を制御するための機能も提供します。

詳細については、「ユーザーズ・ガイド」を参照してください。

IBM 拡張 の終り

Fortran 95 とは何か

Fortran 95

Fortran 95 言語標準には、Fortran 90 と FORTRAN 77 言語標準の機能の大部分が含まれています。 しかしながら、Fortran 90 と FORTRAN 77 言語標準の機能のうちごく一部が Fortran 95 では削除されています。詳細については、 927 ページの『削除された機能』を参照してください。 Fortran 95 標準では、主に次の点が向上しています。

FORALL

Fortran 95 では、Fortran 90 の配列値のエレメント型構造によって、エレメント型の効果的な代替方法を提供しています。 **FORALL** ステートメントを使用すると、配列エレメント、配列セクション、文字サブストリング、またはエレメント型添え字の関数としてのポインター・ターゲットを明示的に指定することができます。

詳細については、 367 ページの『FORALL』を参照してください。

PURE

Fortran 95 関数には、副次作用がある場合があります。発生する可能性がある副次作用には、引き数の値またはグローバル変数の値の変更があります。副次作用に関連した問題を避けるために、Fortran 95 では関数が副次作用の影響を受けないように指定することができます。これらの関数のことを、**PURE** 関数といいます。 **PURE** プロシージャーの制限付き書式のことを **ELEMENTAL** といいます。

詳細については、 209 ページの『純粋プロシージャー』を参照してください。

ELEMENTAL

Fortran 95 では、エレメント型のプロシージャーを用意する代わりに効果的な方法を組み込んでいます。 Fortran 90 では、エレメント型のプロシージャーを用意するために、8 つのバージョンのプロシージャーを作成する必要がありました。つまり、スカラー操作用に 1 つ、1 ～ 7 ランクの配列用にそれぞれ 1 つずつのプロシージャーです。 2 つの配列引き数を使用する場合には、 16 ものバージョンのプロシージャーが必要になります。 **ELEMENTAL** を使用すると、同じ機能を持つ単一のプロシージャーを作成することができます。

詳細については、 212 ページの『エレメント型プロシージャー』を参照してください。

初期化

Fortran 95 は、Fortran 90 からの機能向上として、初期ポインターの関連付け状況を定義する手段を提供しています。 Fortran 95 では、**NULL** 組み込み関数を使用することによって、初期ポインターの関連付け状況が未定義以外に設定されます。

詳細については、 641 ページの『NULL(MOLD)』を参照してください。

また、Fortran 95 には、派生型コンポーネントのデフォルトの初期値を指定する方法が備えられており、Fortran のデータ抽象化機能が拡張されています。

詳細については、 41 ページの『派生型』を参照してください。

仕様関数

Fortran 95 は、Fortran 90 からの機能向上として、宣言式の中にユーザー定義関数を使用できます。仕様関数は、PURE 関数でなければなりません。これにより、同じ *specification_part* に宣言されている他のオブジェクトに副次的な作用が及ぶことがなくなります。

詳細については、 111 ページの『宣言式』を参照してください。

削除された機能

Fortran 95 では、Fortran 90 と FORTRAN 77 言語の標準の機能のうちごく一部が削除されています。 FORTRAN 95 から削除された機能は、Fortran 標準化委員会によって、最新のプログラミングの業務にほとんど準拠していないと見なされたものです。ただし、削除された機能に代わる有効な機能があるので、機能そのものは Fortran 95 から削除されていません。

XL Fortran は、Fortran 95 標準仕様から削除された機能を引き続きサポートしています。

詳細については、 927 ページの『削除された機能』を参照してください。

Fortran 95 の終り

Fortran 90 とは何か

Fortran 90 では、FORTRAN 77 に新たに豊富な機能を追加しています。 Fortran 90 は、特定の作業を実行するために FORTRAN 77 よりも向上した技法または機能を提供しています。 FORTRAN 77 言語に対して Fortran 90 で新たに追加された主な機能の一部について、以下に概要を示します。

Fortran 90 自由ソース形式

固定ソース形式 (FORTRAN 77 で定義されたもの) に加えて、Fortran 90 では、自由ソース形式を定義しています。この形式では、ステートメントはどのカラムからも開始でき、ブランクは有効となります。

注: XL Fortran も IBM の自由ソース形式を定義するため、本書では、Fortran 90 で定義された自由ソース形式を Fortran 90 自由形式と呼びます。

詳細については、 19 ページの『自由ソース形式』を参照してください。

パラメーター化されたデータ型

データ型の長さ指定 (たとえば、INTEGER*4) は、業界共通の拡張機能です。Fortran 90 では、文字以外の組み込みデータ型の精度と範囲、文字のデータ型で利用可能な文字セットを指定する機能を提供しています。 **SELECTED_INT_KIND** および **SELECTED_REAL_KIND** 組み込み関数とともに使用すると、パラメーターを使用するデータ型は、異なるプラットフォーム間で移植可能になります。

詳細については、 27 ページの『型付きパラメーターおよび指定子』を参照してください。

派生型

派生型はユーザー定義型で、そのコンポーネントは組み込み型またはその他の派生型、あるいはその両方です。派生型のオブジェクトは、組み込み機能の割り当て、 I/O、およびプロシージャーの引き数として使用できます。定義または拡張された組み込み操作で、派生型を使用すると、強力なデータ抽出機能が提供されます (たとえば、リンクされたリスト)。

詳細については、 41 ページの『派生型』を参照してください。

配列の機能強化

Fortran 90 では、配列式および割り当てを指定することができます。配列全体の一部である配列セクションを 1 つの配列として使用することができます。配列構造は、配列の値を指定するための簡略化された構文を提供しています。想定形状配列 (assumed-shape arrays)、据え置き形状配列 (deferred-shape arrays) および自動割り振り可能配列 (automatic arrays) によって、配列の使用に柔軟性を持たせています。 **WHERE** 構造体を使用して、配列式の計算および配列の割り当てをマスクしてください。

詳細については、 83 ページの『第 4 章 配列の概念』を参照してください。

ポインター

ポインターは、値ではなくメモリーのアドレスを参照します。ポインターは、リンクされたリストおよび動的配列を作成する手段を提供しています。組み込み型または派生型のオブジェクトをポインターとして宣言することができます。

詳細については、 167 ページの『ポインター関連付け』および 431 ページの『POINTER (Fortran 90)』を参照してください。

動的処理

ストレージは、コンパイル時にポインターのターゲットおよび割り振り可能配列のために確保されません。 **ALLOCATE** および **DEALLOCATE** ステートメントを使用して、実行時にストレージの使用方法を制御することができます。また、ポインター割り当てを使用して、ポインターに関連する記憶スペースを変更することもできます。

詳細については、 290 ページの『ALLOCATE』、 331 ページの『DEALLOCATE』、および 142 ページの『ポインターの割り当て』を参照してください。

制御構造体の機能強化

CASE 構造体では、実行するステートメント・ブロックの中から 1 つのみを選択するための簡略化された構文を提供しています。各 **CASE** ブロックのケース式は、構造体のケース式に照らして計算されます。

制御文節なしの **DO** ステートメントおよび **DO WHILE** 構造体は、その汎用性が高められています。さらに、**CYCLE** および **EXIT** ステートメントは、構造体内から実行される構造体を制御します。

制御構造体には名前を付けることができますが、これはネスト構造体で読みやすさを向上させ、構文チェックを行いやすくします。

詳細については、 147 ページの『第 6 章 制御構造』を参照してください。

プロシージャーの機能強化

Fortran 90 では、プロシージャーを容易に使用するために多くの新しい機能を取り入れています。関数を使用して、組み込み演算子を拡張したり、新しい演算子を定義することができます。サブルーチンにより、組み込みの割り当てを拡張することができます。プロシージャーの実引き数をキーワードで指定することができ、仮引き数の意図的な使用やオプションであるか否かについて明示的に示すことができます。

仮プロシージャーまたは外部プロシージャーのインターフェースや特性をインターフェース・ブロックに明示することができます。汎用インターフェース・ブロックでは、実引き数の性質に基づいて、ブロック内で定義した特定のプロシージャーのいずれかにアクセスするために、参照する名前を指定することができます。

メインプログラムや他のサブプログラムに含まれる内部プロシージャーにより、ホスト・プロシージャーで定義されたエンティティーにアクセスする一方で、プログラムを区分することができます。

Fortran 90 では再帰的プロシージャーが可能です。関数およびサブルーチンは、直接または間接的に自らを呼び出すことができます。

詳細については、 159 ページの『第 7 章 プログラム単位およびプロシージャー』を参照してください。

モジュール

モジュールは、データのカプセル化の手段およびデータに適用する操作を提供します。モジュールとは、実行不能プログラム単位のことで、これにはデータ・オブジェクトの宣言、派生型の定義、プロシージャー、およびプロシージャー・インターフェースを含めることができます。モジュールを使うと、あるエンティティーは特定のモジュール内でのみ使用して、その他のエンティティーは任意のプログラム単位からアクセスできるように指定することもできます。

詳細については、 183 ページの『モジュール』を参照してください。

新規の組み込みプロシージャー

Fortran 90 では、Fortran に対して新たに多くの組み込みプロシージャーを追加しています。たとえば、ある変形組み込みプロシージャーでは、強力な配列操作機能を提供します。新規に追加された豊富な照会機能により、エンティティーの特性を調べることができます。

詳細については、 539 ページの『第 12 章 組み込みプロシージャー』を参照してください。

有効な XL Fortran プログラムと無効な XL Fortran プログラム

本書では、有効な XL Fortran プログラムを作成する際に従わなければならない構文、セマンティクス、および制約を定義します。 XL Fortran 言語規則に対するほとんどの違反はコンパイラーによって検出されますが、コンパイラーによっては検出されない構文やセマンティクスの組み合わせもあります。これは、パフォーマンス上の理由か、違反が実行時にしか検出できないことなどが理由です。こうした診断されない組み合わせを含む XL Fortran プログラムは、そのプログラムが期待どおりに実行されるかどうかに関係なく、無効なプログラムとなります。

第 2 章 言語エレメント

この章では、XL Fortran プログラムのエレメントを説明します。

- 『文字』
- 12 ページの『名前』
- 13 ページの『ステートメント』
- 14 ページの『行およびソース形式』
- 24 ページの『ステートメントおよび実行の順序』

文字

XL Fortran の文字セットは、文字、数字、特殊文字で構成されています。

文字	数字	特殊文字
A N * a n	0	ブランク
B O b o	1	= 等号
C P c p	2	+ 正符号
D Q d q	3	- 負符号
E R e r	4	* アスタリスク
F S f s	5	/ スラッシュ
G T g t	6	(左括弧
H U h u	7	) 右括弧
I V i v	8	, コンマ
J W j w	9	. 小数点/ピリオド
K X k x		$ ドル記号
L Y l y		' アポストロフィ
M Z m z *		: コロン
		! 感嘆符
		" 二重引用符
		% パーセント記号
		& アンパーサンド
		; セミコロン
		? 疑問符
		< より小
		> より大
		_ 下線

IBM 拡張

注: * XL Fortran では、英小文字が使用されます。

IBM 拡張 の終り

これらの文字は、*照合順序* と呼ばれる順序を持っています。照合順序とは、処理 (ソート、組み合わせ、比較など) の一連の順序の判別基準となる文字の配列です。 XL Fortran では、ASCII (情報交換用米国標準コード) を使用して、通常の文字順序を決定しています。 (ASCII 文字セットの完全なリストについては、 929 ページの『付録 B. ASCII 文字セットと EBCDIC 文字セット』を参照してください。)

ホワイト・スペース とは、ブランクとタブのことです。ホワイト・スペースの意味は、使用するソース形式によって決まります。詳細については、 14 ページの『行およびソース形式』を参照してください。

字句トークン とは、分割しないで解釈する文字列のことで、これによってプログラムの構築ブロックが形成されます。これには、キーワード、名前、リテラル定数 (複素数タイプ以外のもの)、演算子、ラベル、区切り文字、コンマ、等号、コロン、セミコロン、パーセント記号、**::**、**=>** などがあります。

名前

名前 は、以下のエレメントのいくつか、またはすべてを並べて構成されます。
- 文字 (A 〜 Z、a 〜 z)
- 数字 (0-9)
- 下線 (_)
- ▶ IBM ドル記号 ($) IBM ◀

名前の最初の文字は数字以外でなければなりません。

Fortran 90 および Fortran 95 では、名前の最大長は 31 文字です。

IBM 拡張

XL Fortran では、名前の最大長は 250 文字です。 XL Fortran では名前を下線で始めることもできますが、 AIX® オペレーティング・システム、および XL Fortran コンパイラーとライブラリーなどの予約名は下線で始まるため、名前の先頭には下線を使用しないことをお勧めします。

ソース・プログラム内の英字は、文字コンテキストの中のものを除いて、すべて小文字に変換されます。文字コンテキストとは、文字リテラル定数、文字ストリング編集記述子、およびホレリス定数内の文字です。

注: -qmixed コンパイラー・オプションを指定した場合、名前は小文字に変換されません。たとえば、XL Fortran では、

```
ia Ia iA IA
```

は、デフォルトではすべて同じものとして扱われます。ただし、**-qmixed** コンパイラー・オプションを指定すると、別の名前として扱われます。

IBM 拡張 の終り

名前は、以下のエンティティーを識別します。
- 変数
- 定数
- プロシージャー
- 派生型
- 構造体
- **CRITICAL** 構造体
- プログラム単位
- 共通ブロック
- 名前リスト・グループ

サブオブジェクト指定子は、その後に 1 つ以上のセレクター (配列エレメント・セレクター、配列セクション・セレクター、コンポーネント・セレクター、サブストリング・セレクター) が続く名前のことです。これは、プログラム単位内の以下の項目を識別します。
- 配列エレメント (94 ページの『配列エレメント』を参照)
- 配列セクション (95 ページの『配列セクション』を参照)
- 構造体コンポーネント (48 ページの『構造体コンポーネント』を参照)
- 文字サブストリング (40 ページの『文字サブストリング』を参照)

ステートメント

Fortran ステートメントとは、字句トークンが連続したものです。ステートメントによってプログラム単位が構成されます。

IBM 拡張

XL Fortran では、ステートメントの最大長は 6700 文字です。

IBM 拡張 の終り

XL Fortran でサポートされるステートメントの詳細については、『ステートメントと属性』を参照してください。

XL Fortran でサポートされるステートメントの詳細については、 285 ページの『第 10 章 ステートメントおよび属性』を参照してください。

ステートメント・キーワード

ステートメント・キーワードはステートメントの構文の一部をなし、すべての箇所 (構文図および表を除く) で太字の大文字で示されます。たとえば、**DATA** ステートメントの **DATA** という用語はステートメント・キーワードです。

コンテキストで予約されている文字ストリングはありません。ステートメントのキーワードがこのようなコンテキストで使用されている場合は、そのキーワードをエンティティー名として解釈します。

ステートメント・ラベル

ステートメント・ラベルは、1 から 5 桁の数字の列で、そのうちの少なくとも 1 桁はゼロ以外の数字でなければなりません。このラベルは、Fortran の有効範囲単位内のステートメントを識別するために使用します。固定ソース形式のステートメントの場合、ステートメントの開始行の 1 ～ 5 桁までのいずれかにステートメント・ラベルを付けることができます。自由ソース形式の場合は、こうした桁についての制約事項はありません。

IBM 拡張

XL Fortran は、固定ソース形式の継続行では 1 ～ 5 桁目までに示されるすべての文字を無視します。

IBM 拡張 の終り

1 つの有効範囲単位内で複数のステートメントに同じラベルを指定すると、あいまいさが生じ、コンパイラーがエラーを生成します。ホワイト・スペースおよび先行ゼロは、ステートメント・ラベルの識別においては意味を持ちません。どのステートメントにもラベルは指定できますが、ステートメント・ラベルで参照できるのは、実行可能ステートメントと **FORMAT** ステートメントに限られます。参照が行われるようにするため、参照するステートメントと参照されるステートメント (ステートメント・ラベルによって識別される) は、同一の有効範囲単位内になければなりません。 (詳細については、159 ページの『有効範囲』を参照してください。)

行およびソース形式

行とは、水平方向の文字の列です。これに対して、桁は垂直方向の文字の列で、特定の桁にあるそれぞれの文字 (マルチバイト文字の場合は各バイト) の位置は行内で同じになります。

IBM 拡張

XL Fortran は、行の長さをバイト単位で表すので、これらの定義が適用されるのは 1 バイト文字を含む行だけです。マルチバイト文字の場合は、各バイトが 1 桁を占めま

す。

IBM 拡張 の終り

行には以下の種類があります。

開始行	ステートメントの先頭の行です。
継続行	開始行の次行以降にステートメントを継続させる行。
注釈行	実行可能プログラムに影響を与えないので、説明の記入に使用することができます。注釈テキストは行の終わりまで継続します。複数の注釈行を次々に継続させていくことはできますが、1 つの注釈行を複数の行に渡って継続させることはできません。行全体がホワイト・スペースの行または長さがゼロの行は、テキストのない注釈行と見なされます。注釈テキストには、文字コンテキストで使用できる文字であれば、どの文字でも入れることができます。 開始行または継続行を継続させない場合、あるいは開始行または継続行を継続させるがその継続が文字コンテキスト内で行われない場合、同じ行で、ステートメント・ラベル、ステートメント・テキスト、および継続文字のいずれかに続けてインライン注釈を入れることができます。感嘆符 (!) はインライン注釈の始まりを意味します。
*** 条件付きコンパイル行**	行がコンパイルされるのは、条件付きコンパイル行の認識が使用可能になっている場合だけであることを示します。条件付きコンパイル標識は、条件付きコンパイル行になければなりません。 (22 ページの『条件付きコンパイル』を参照してください。) *****
*** デバッグ行**	その行がデバッグ・コード用であることを示します (固定ソース形式の場合のみ)。 XL Fortran では、1 桁目に `D` または `X` の文字が指定されていなければなりません。 (18 ページの『デバッグ行』を参照してください。) *****
*** ディレクティブ行**	XL Fortran では、コンパイラーに指示または情報を与えます (507 ページの『第 11 章 汎用ディレクティブ』を参照してください)。*****

IBM 拡張

注: * XL Fortran では、デバッグ行またはディレクティブ行が使用されます。

XL Fortran では、ソース入力行は、固定形式と自由形式のどちらのソース形式でもかまいません。同一のプログラム単位内でソース形式を混在させるには、 **SOURCEFORM** ディレクティブを使用します。 **f77**、**fort77**、**xlf**、**xlf_r**、または **xlf_r7** 呼び出しコマンドを使用する場合は、固定ソース形式がデフォルトになります。 **xlf90**、**xlf90_r**、**xlf90_r7**、**xlf95**、**xlf95_r**、または **xlf95_r7** 呼び出しコマンドを使用する場合には、Fortran 90 自由ソース形式がデフォルトになります。

呼び出しコマンドの詳細については、「ユーザーズ・ガイド」の『*XL Fortran プログラムのコンパイル*』を参照してください。

IBM 拡張 の終り

固定ソース形式

IBM 拡張

固定ソース形式の行は、1 〜 132 文字の文字列です。デフォルトの行サイズは (Fortran 95 に規定されているとおり) 72 文字ですが、 XL Fortran では、**-qfixed**=*right_margin* コンパイラー・オプションを使用して変更することができます (「ユーザーズ・ガイド」を参照してください)。

IBM 拡張 の終り

右マージンを超える行は、行の一部ではないので、識別、番号付け、またはその他の目的に使用できます。

文字コンテキスト内でなければ、ホワイト・スペースは意味を持ちません。つまり、ホワイト・スペースを字句トークン内または字句トークンの間に埋め込むことができます。このような使い方をしてもコンパイラーのそれらの字句トークンの扱い方に影響はありません。

IBM 拡張

タブ形式設定では、 XL Fortran の開始行の 1 〜 6 行目にタブ文字があることを意味します。これは、タブ文字の次の文字を 7 桁目から始めるように、コンパイラーに指示します。

IBM 拡張 の終り

固定ソース形式の行における行および項目の要件は以下のとおりです。

- 注釈行は、C、c、または 1 桁目のアスタリスク (*) で始まるか、またはすべてホワイト・スペースです。注釈は、感嘆符 (!) に続けて記入することもできます。ただし、感嘆符が 6 桁目または文字コンテキスト中にある場合を除きます。
- タブが形式設定されていない開始行の場合
  – 1 〜 5 桁目に、ブランクが入るか、ステートメント・ラベルが入るか、または 1 桁目に D または X が入り、その後にオプションでステートメント・ラベルが続きます。
  – 6 桁目にブランクまたはゼロが入ります。
  – 7 桁目から右マージンまで、ステートメント・テキストが入ります。その後に、他のステートメントまたはインライン注釈が続くこともあります。

IBM 拡張

- XL Fortran で、タブが形式設定されている開始行の場合:
  - 1 ～ 6 桁目に、ブランクが入るか、ステートメント・ラベルが入るか、または 1 桁目に D または X が入り、その後にオプションでステートメント・ラベルが続きます。この後にはタブ文字が必要です。
  - **-qxflag=oldtab** コンパイラー・オプションを指定した場合は、タブ文字の直後の桁から右マージンまでのすべての桁にステートメント・テキストが入ります。その後に他のステートメントやインライン注釈が続くこともあります。
  - **-qxflag=oldtab** コンパイラー・オプションを指定していない場合は、 7 桁目 (タブの後の文字に相当するもの) から右マージンまでのすべての桁にステートメント・テキストが入ります。その後に他のステートメントやインライン注釈が続くこともあります。

IBM 拡張 の終り

- 継続行の場合
  - 1 桁目には、C、c、またはアスタリスクを入れることはできません。 1 ～ 5 桁目には、最初の非ブランク文字として感嘆符を入れることはできません。

IBM 拡張

XL Fortran では、1 桁目に D (デバッグ行を示す) を入れることができます。 D を入れない場合には、1 ～ 5 桁目には文字コンテキスト内で使用できるどのような文字でも入れることができますが、その文字は無視されます。

IBM 拡張 の終り

  - 6 桁目の文字は、ゼロ以外の文字またはホワイト・スペース以外の文字のいずれかにしなければなりません。 6 桁目の文字は、継続文字と呼ばれます。感嘆符およびセミコロンは有効な継続文字です。
  - 7 桁目から右マージンまでに継続ステートメントが入ります。その後に、他のステートメントおよびインライン注釈が続くこともあります。
  - **END** ステートメント、およびステートメントの先頭行にプログラム単位の **END** ステートメントがあるステートメントは、どちらも継続できません。

IBM 拡張

  - XL Fortran では、ステートメントの継続行の数に制約はありません。ただし、ステートメントの文字数が 6700 文字を超えることはできません。

IBM 拡張 の終り

Fortran 標準では、継続行の数を 19 までに制限しています。

セミコロン (;) は、文字コンテキストの中、注釈の中、または 1 〜 6 桁目に使用した場合以外は、単一ソース行上のステートメントを区切る役割を果たします。同じ行に 2 つ以上のセミコロン・セパレーターがあり、それらがホワイト・スペースまたは別のセミコロンで区切られている場合は、単一のセパレーターと見なされます。セパレーターが行の最後にある場合やインライン注釈の前にある場合は、そのセパレーターは無視されます。同じ行の中で、セミコロンに続くステートメントに、ラベルを付けることはできません。同じ行の中で、追加のステートメントをプログラム単位の **END** ステートメントに続けることはできません。

デバッグ行

IBM 拡張

デバッグ行は、固定ソース形式のみが使用可能であり、デバッグ用に使用されるソース・コードが入っています。このデバッグ行は、XL Fortran では、文字 D または 1 桁目の文字 X によって指定されます。デバッグ行の処理は、コンパイラー・オプション **-qdlines** または **-qxlines** によって異なります。

- **-qdlines** オプションを指定すると、コンパイラーは 1 桁目の D をブランクと見なし、デバッグ行をソース・コード行として扱います。 **-qxlines** を指定する場合は、コンパイラーは、1 桁目の X をブランクとして解釈し、これらの行をソース・コードとして扱います。
- **-qdlines** または **-qxlines** オプションを指定しなければ、コンパイラーはデバッグ行を注釈行として扱います。これはデフォルト設定です。

デバッグ・ステートメントが複数の行にまたがっている場合は、継続行の 1 桁目に D または X のような継続文字を指定しなければなりません。開始行がデバッグ行でない場合でも、継続行をデバッグ行として指定することができます。ただし、**-qdlines** または **-qxlines** コンパイラー・オプションを指定したかどうかにかかわらず、ステートメントの構文は正しくなければなりません。

IBM 拡張 の終り

固定ソース形式の例:

```
C Column Numbers:
C       1         2         3         4         5         6         7
C23456789012345678901234567890123456789012345678901234567890123456789012

!IBM* SOURCEFORM (FIXED)
      CHARACTER ABC ; LOGICAL X          ! 2 statements on 1 line
      DO 10 I=1,10
        PRINT *,'this is the index',I  ! with an inline comment
10    CONTINUE
C
       CHARSTR="THIS IS A CONTINUED
     X CHARACTER STRING"
```

```
      ! There will be 38 blanks in the string between "CONTINUED"
      ! and "CHARACTER". You cannot have an inline comment on
      ! the initial line because it would be interpreted as part
      ! of CHARSTR (character context).
  100 PRINT *, IERROR
! The following debug lines are compiled as source lines if
! you use -qdlines
D     IF (I.EQ.IDEBUG.AND.
D    +    J.EQ.IDEBUG)     WRITE(6,*) IERROR
D     IF (I.EQ.
D    +  IDEBUG )
D    +  WRITE(6,*) INFO
      END
```

自由ソース形式

IBM 拡張

XL Fortran では、文字数が 6700 を超えなければ、行の長さや継続行の数を任意に指定することができます。

IBM 拡張 の終り

自由ソース形式の行では、1 行に 132 文字まで指定することができ、1 ステートメントに最大で 39 までの継続行が指定できます。

項目は行の任意の桁位置から始めることができ、行および行に関する項目の次の要件を満たしています。

- 注釈行は、ホワイト・スペースだけで構成される行、または感嘆符 (!) で始まる行です。ただし、感嘆符が文字コンテキストの一部である場合を除きます。
- 開始行には、次の項目のいずれかが以下の順で入っています。
  - ステートメント・ラベル。
  - ステートメント・テキスト。開始行には、ステートメント・テキストが必須であることに注意してください。
  - 追加ステートメント
  - アンパーサンド継続文字 (&)
  - インライン注釈
- 開始行または継続行を非文字コンテキストで継続させる場合、継続行は、開始行または継続行に続く最初の注釈以外の行から始まるようにする必要があります。行を継続行として定義するには、直前の注釈以外の行のステートメントの後ろにアンパーサンドを置く必要があります。
- アンパーサンドの前後のホワイト・スペースは任意ですが、次の制約事項があります。

– 継続行の最初の非ブランク文字位置にアンパーサンドを置く場合、ステートメントはアンパーサンドに続く次の文字位置から継続します。
– 字句トークンを継続させる場合、トークンの前半部分の直後にアンパーサンドを続け、継続行でアンパーサンドの直後にトークンの後半部分を続けなければなりません。

- 文字コンテキストは、次の条件が真の場合に継続します。
  – 継続行の最後の文字がアンパーサンドで、その後にインライン注釈が続かない。継続させるステートメント・テキストの最後の文字がアンパーサンドの場合、継続文字としてもう 1 つのアンパーサンドを入力しなければなりません。
  – 次の非注釈行の先頭の非ブランク文字がアンパーサンドである。

単一ソース行の複数のステートメントは、セミコロンで区切ります。ただしセミコロンが文字コンテキスト中または注釈に現れる場合を除きます。同じ行に 2 つ以上のセパレーターがあり、それらがホワイト・スペースまたは別のセミコロンで区切られている場合は、単一のセパレーターと見なされます。セパレーターが行の最後にある場合やインライン注釈の前にある場合は、そのセパレーターは無視されます。同じ行の中で、追加のステートメントをプログラム単位の **END** ステートメントに続けることはできません。

ホワイト・スペース

ホワイト・スペースは、字句トークン内に入れることはできません。ただし文字コンテキスト内またはフォーマット指定内は除きます。ホワイト・スペースは、読みやすさを向上させるために、トークンの間に自由に挿入できます。ただし、名前、定数、ラベルを、隣接するキーワード、名前、定数、ラベルから区切るものでなければなりません。

隣接する特定のキーワード間でホワイト・スペースが必要となる場合があります。以下の表では、ホワイト・スペースが必須であるキーワード、およびホワイト・スペースがオプションであるキーワードをリストしています。

表 1. ホワイト・スペースがオプションであるキーワード

BLOCK DATA	END FUNCTION	END SUBROUTINE
DOUBLE COMPLEX	END IF	END TYPE
DOUBLE PRECISION	END INTERFACE	END UNION
ELSE IF	END MAP	END WHERE
END BLOCK DATA	END MODULE	GO TO
END DO	END PROGRAM	IN OUT
END FILE	END SELECT	SELECT CASE
END FORALL	END STRUCTURE	

type_spec の詳細については、 481 ページの『タイプ宣言』を参照してください。

自由ソース形式の例:

```
!IBM* SOURCEFORM (FREE(F90))
!
! Column Numbers:
!        1         2         3         4         5         6         7
!23456789012345678901234567890123456789012345678901234567890123456789012
 DO I=1,20
   PRINT *,'this statement&
   & is continued' ; IF (I.LT.5) PRINT *, I

 ENDDO
 EN&
        &D              ! A lexical token can be continued
```

IBM 自由ソース形式

IBM 拡張

IBM 自由ソース形式の行またはステートメントは、最大で 6700 文字までの文字列です。項目は行の任意の桁位置から始めることができ、行および行に関する項目の次の要件を満たしています。

- 注釈行は、1 桁目が二重引用符 (") で始まり、ホワイト・スペースだけで構成される行またはゼロ長の行です。注釈行は、継続行の後に続けてはなりません。注釈は、感嘆符(!) に続けて記入することもできます。ただし、感嘆符が文字コンテキストの一部である場合を除きます。
- 開始行には、次の項目のいずれかが以下の順で入っています。
  - ステートメント・ラベル
  - ステートメント・テキスト
  - 負符号継続文字 (-)
  - インライン注釈
- 継続行は、継続する行のすぐ後に続けられ、次の項目のいずれかが以下の順で入ります。
  - ステートメント・テキスト
  - 継続文字 (-)
  - インライン注釈

開始行または継続行上のステートメント・テキストを継続させる場合、負符号を使用して、ステートメント・テキストが次の行に継続することを示します。文字コンテキスト中で、継続させるステートメント・テキストの最後の文字が負符号の場合、継続文字としてもう 1 つの負符号を入力する必要があります。

文字コンテキスト内でなければ、ホワイト・スペースは意味を持ちません。つまり、ホワイト・スペースを字句トークン内または字句トークンの間に埋め込むことができます。このような使い方をしてもコンパイラーのそれらの字句トークンの扱い方に影響はありません。

IBM 自由ソース形式の例

```
!IBM* SOURCEFORM (FREE(IBM))
"
" Column Numbers:
"        1         2         3         4         5         6         7
"23456789012345678901234567890123456789012345678901234567890123456789012
 DO I=1,10
  PRINT *,'this is -
             the index',I   ! There will be 14 blanks in the string
                            ! between "is" and "the"
 END DO
 END
```

IBM 拡張 の終り

条件付きコンパイル

IBM 拡張

XL Fortran プログラムの特定の行を条件付きコンパイルするようマークするには、標識を使用します。 このサポートを使用すれば、SMP 環境内だけで有効なステートメントまたは SMP 環境内だけで必要なステートメントが入っているコードを、非 SMP 環境に移植することができます。これを行うには、条件付きコンパイル行を使用するか、**_OPENMP** C プリプロセッサー・マクロを使用します。

条件付きコンパイル行の構文は、次のとおりです。

```
►►─cond_comp_sentinel─fortran_source_line─────────────────────────►◄
```

cond_comp_sentinel
: 現行ソース形式によって定義される条件付きコンパイル標識です。これは、次のいずれかとなります。
  - 固定ソース形式の場合は、**!$**、**C$**、**c$**、または ***$**。
  - 自由ソース形式の場合は、**!$**。

fortran_source_line
: XL Fortran ソース行です。

条件付きコンパイル行の構文規則は、固定ソース形式行と自由ソース形式行の構文規則とよく似ています。構文規則は、次のとおりです。

- 一般規則:

  有効な XL Fortran ソース行が、条件付きコンパイル標識に続いていなければなりません。

条件付きコンパイル行には、**INCLUDE** 非注釈ディレクティブまたは **EJECT** 非注釈ディレクティブが入っていても構いません。

条件付きコンパイル標識には、組み込みホワイト・スペースが入っていてはなりません。

条件付きコンパイル標識は、同一行のソース・ステートメントまたはディレクティブの後に置くことはできません。

条件付きコンパイル行を継続させる場合は、条件付きコンパイル標識が、少なくとも 1 つの継続行か先頭の行になければなりません。

条件付きコンパイル行が認識されるようにするには、 **-qcclines** コンパイラー・オプションを指定する必要があります。条件付きコンパイル行が認識されないようにするには、 **-qnocclines** コンパイラー・オプションを指定します。 **-qsmp=omp** コンパイラー・オプションを指定すると、 **-qcclines** オプションを指定できるようになります。

トリガー・ディレクティブは、条件付きコンパイル標識に優先します。たとえば、**-qdirective='$'** オプションを指定すると、 **!$** などのトリガーで始まる行は、条件付きコンパイル行ではなく注釈ディレクティブとして扱われます。

- 固定ソース形式の規則:

  条件付きコンパイル標識は、1 桁目から始まっていなければなりません。

  固定ソース形式行の長さ、大文字小文字の区別、ホワイト・スペース、継続、タブ形式設定、および桁についてのすべての規則が適用されます。詳細については、 16 ページの『固定ソース形式』を参照してください。条件付きコンパイル行が認識されるようになる場合は、条件付きコンパイル標識は 2 つのホワイト・スペースによって置き換えられます。

- 自由ソース形式の規則:

  条件付きコンパイル標識は、どの桁から始まっていても構いません。

  自由ソース形式行の長さ、大文字小文字の区別、ホワイト・スペース、継続についてのすべての規則が適用されます。詳細については、 19 ページの『自由ソース形式』を参照してください。条件付きコンパイル行が認識されるようになる場合は、条件付きコンパイル標識は 2 つのホワイト・スペースによって置き換えられます。

コードを条件付きで組み込むための別の方法 (条件付きコンパイル行を使用する方法以外) は、C プリプロセッサー・マクロ **_OPENMP** を使用する方法です。このマクロが定義されるのは、C プリプロセッサーが呼び出されて、 **-qsmp=omp** コンパイラー・オプションを指定した時です。このマクロの使用方法の例については、「ユーザーズ・ガイド」の 『XL Fortran プログラムの編集、コンパイル、リンク、実行』の中の『C プリプロセッサーによる Fortran ファイルの引き渡し』を参照してください。

有効な条件付きコンパイル行の例

次の例では、条件付きコンパイル行が、 OpenMP 実行時ルーチンを隠すために使用されています。 OpenMP 実行時ルーチンを呼び出すコードは、非 OpenMP 環境では条件

付きコンパイルを使用しないで簡単にコンパイルすることはできません。実行時ルーチンに対する呼び出しはディレクティブではないので、これらの呼び出しを !$OMP トリガーによって隠すことはできません。以下のコードを **-qsmp=omp** コンパイラー・オプションを指定せずにコンパイルすると、スレッドの数を保管するために使用される変数には、値 8 が割り当てられます。

```
      PROGRAM PAR_MAT_MUL
      IMPLICIT NONE
      INTEGER(KIND=8)                  ::I,J,NTHREADS
      INTEGER(KIND=8),PARAMETER        ::N=60
      INTEGER(KIND=8),DIMENSION(N,N)   ::AI,BI,CI
      INTEGER(KIND=8)                  ::SUMI
!$    INTEGER OMP_GET_NUM_THREADS

      COMMON/DATA/ AI,BI,CI
!$OMP THREADPRIVATE (/DATA/)

!$OMP PARALLEL
      FORALL(I=1:N,J=1:N) AI(I,J) = (I-N/2)**2+(J+N/2)
      FORALL(I=1:N,J=1:N) BI(I,J) = 3-((I/2)+(J-N/2)**2)
!$OMP MASTER
      NTHREADS=8
!$    NTHREADS=OMP_GET_NUM_THREADS()
!$OMP END MASTER
!$OMP END PARALLEL

!$OMP PARALLEL DEFAULT(PRIVATE),COPYIN(AI,BI),SHARED(NTHREADS)
!$OMP DO
      DO I=1,NTHREADS
      CALL IMAT_MUL(SUMI)
      ENDDO
!$OMP END DO
!$OMP END PARALLEL

      END
```

IBM 拡張 の終り

ステートメントおよび実行の順序

表 2. ステートメントの順序

1 PROGRAM、FUNCTION、SUBROUTINE、MODULE、BLOCK DATA のいずれかのステートメント
2 USE ステートメント

表 2. ステートメントの順序 (続き)

<table>
<tr><td rowspan="2">3 DATA、FORMAT、および ENTRY ステートメント</td><td>4 派生型定義、インターフェース・ブロック、タイプ宣言ステートメント、仕様ステートメント、IMPLICIT ステートメント、PARAMETER ステートメント</td></tr>
<tr><td>5 実行可能構造体</td></tr>
<tr><td colspan="2">6 CONTAINS ステートメント</td></tr>
<tr><td colspan="2">7 内部サブプログラムまたはモジュール・サブプログラム</td></tr>
<tr><td colspan="2">8 END ステートメント</td></tr>
<tr><td colspan="2">ステートメントの順序

縦線の範囲内では、さまざまなステートメントを選択することができますが、水平線を超えてステートメントを入れ換えることはできません。図中の数字は、特定のコンテキストで使用できる一群のステートメントを識別するために本書の中で後ほど使用されます。この節の参照は、本書の他の箇所でこれらの番号が使用される場所に記載されています。</td></tr>
</table>

ステートメントの順序に関する規則および制限の詳細については、 159 ページの『第 7 章 プログラム単位およびプロシージャー』または 285 ページの『第 10 章 ステートメントおよび属性』を参照してください。

通常の実行順序は、指定した関数への参照が任意の順序で処理され、それに続いて実行可能なステートメントが有効範囲単位に現れる順番で処理されます。

制御の転送は、通常の実行順序の代わりとなるものです。実行順序を制御するために使用できるステートメントとして以下のものがあります。

- 制御ステートメント
- **END=**、**ERR=**、**EOR=** 指定子のいずれかを含む I/O ステートメント

サブプログラムによって定義されているプロシージャーを参照する場合、プログラムの実行は、プロシージャーを定義しているサブプログラムの有効範囲単位で参照される、指定された関数で継続します。プログラムは、プロシージャーを定義している **FUNCTION**、**SUBROUTINE**、または **ENTRY** ステートメントに続く最初の実行可能ステートメントで再開されます。サブプログラムから戻ると、プログラムの実行は、プロシージャーが参照された場所、または代替戻り指定子によって参照されるステートメントから継続されます。

本書では、特定の制御の転送でのイベントの順序に関する記述は、エラーの発生や **STOP** ステートメントの実行などのイベントによって通常の順序が変更されないことを前提としています。

第 3 章 データ型およびデータ・オブジェクト

この章では、以下の項目について説明します。

- 『データ型』
- 28 ページの『データ・オブジェクト』
- 29 ページの『組み込み型』
- 41 ページの『派生型』
- IBM 63 ページの『タイプなしリテラル定数』 IBM
- 69 ページの『タイプの決め方』
- 70 ページの『変数の定義状況』
- 77 ページの『割り振り状況』
- 78 ページの『変数のストレージ・クラス』

データ型

データ型には、名前、有効な値、それらの値 (定数) を表す手段、およびそれらの値を操作するための演算が含まれます。データ型は、*組み込み型* と*派生型* の 2 種類に分類されます。

組み込み型は、その演算も含めて事前定義されており、常にアクセスできます。組み込みデータ型には、次の 2 種類があります。

- **数値 (算術とも呼ばれる) :** 整数、実数、複素数、バイト
- **非数値:** 文字、論理、およびバイト

派生型は、ユーザー定義のデータ型で、そのコンポーネントは組み込みまたは派生データ型の一方または両方です。

型付きパラメーターおよび指定子

XL Fortran は、個々の組み込みデータ型について 1 つ以上の表現方法を提供しています。それぞれの方法は *kind 型付きパラメーター* と呼ばれる値によって指定できます。この値は、整数タイプでは 10 進の指数の範囲、実数タイプと複素数タイプでは精度と指数の範囲を指定し、文字タイプと論理タイプでは表現手段を指定します。それぞれの組み込み型は、特定の kind 型付きパラメーターをサポートしています。*kind_param* は、*数字ストリング* または*スカラー整数定数名* のいずれかです。

length 型付きパラメーター は、型付き文字のエンティティーの文字数を指定します。

タイプ指定子 は、タイプ宣言ステートメントで宣言されたすべてのエンティティーのタイプを指定します。タイプ指定子 (**INTEGER**、 **REAL**、**COMPLEX**、**LOGICAL**、**CHARACTER**) の中には、 *kind_selector* を含むことのできるものもあります。この *kind_selector* は、*kind* 型付きパラメーターを指定します。

IBM 拡張

たとえば、4 バイトの整数は、 **INTEGER(4)**、**INTEGER(KIND=4)**、**INTEGER*4** のいずれかとして宣言することができます。あるいは、デフォルトの整数サイズが 4 バイトに設定されている場合、単に **INTEGER** と宣言することができます。 本書では、4 バイト整数の参照について、**INTEGER(4)** の形式をとります。 タイプ指定子の使用に関する詳細については、 481ページの *type_spec* を参照してください。

IBM 拡張 の終り

KIND 組み込み関数は、その引き数の kind 型付きパラメーターを戻します。詳細については、 609 ページの『KIND (X)』を参照してください。

データ・オブジェクト

データ・オブジェクト は、変数、定数、定数のサブオブジェクトのいずれかです。

変数 は値を持つことができ、実行可能プログラムの実行時に定義または再定義することができます。変数には次のものがあります。
- スカラー変数名
- 配列変数名
- サブオブジェクト

変数のサブオブジェクト とは、参照したり定義したりすることが可能な名前付きのオブジェクトの一部です。サブオブジェクトには次のものがあります。
- 配列エレメント
- 配列セクション
- 文字サブストリング
- 構造体コンポーネント

定数のサブオブジェクトは、定数の一部です。参照される部分は、変数の値により異なります。

定数

定数 は値を持ち、実行可能プログラムの実行時に定義または再定義することはできません。名前の付いた定数を*名前付き定数* といいます (429 ページの『PARAMETER』を参照)。名前の付いていない定数は、*リテラル定数* といいます。リテラル定数は、組み込み型であっても、タイプなし (16 進数、8 進数、2 進数、またはホレリス) であっても

かまいません。リテラル定数のオプションの kind 型付きパラメーターは、数字ストリングまたはスカラー整数の名前付きの定数です。

符号付きのリテラル定数では、正符号または負符号が先行します。その他のリテラル定数はすべて、符号なしでなければなりません (先行する符号があってはなりません)。ゼロの値は、正または負のいずれにも見なされません。ゼロは、符号付きまたは符号なしのいずれとしても指定できます。

自動オブジェクト

自動オブジェクト とは、プロシージャー内で動的に割り振られるデータ・オブジェクトです。これは、サブプログラムのローカル・エンティティーで、非定数の文字長および非定数の配列境界の一方またはその両方です。これは仮引き数ではありません。

自動オブジェクトは、制御された自動ストレージ・クラスを常に持っています。

自動オブジェクトは、**DATA**、**EQUIVALENCE**、**NAMELIST**、**COMMON** のステートメント内で指定することはできません。また、 **AUTOMATIC**、 **STATIC**、 **PARAMETER**、 **SAVE** の属性をそれに指定することもできません。自動オブジェクトは、タイプ宣言ステートメント内で初期化式によって初期化したり定義したりすることはできません。ただし、デフォルトの初期化は可能です。 また、メインプログラムまたはモジュールの指定部分に入れることもできません。

組み込み型

整数

IBM 拡張

次の表では、 XL Fortran が整数データ型を使用して表すことのできる値の範囲を示しています。

Kind パラメーター	値の範囲
1	-128 〜 127
2	-32 768 〜 32 767
4	-2 147 483 648 〜 2 147 483 647
8	-9 223 372 036 854 775 808 〜 9 223 372 036 854 775 807

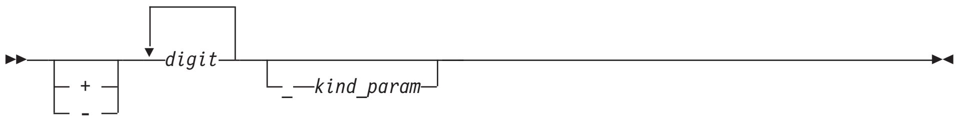
XL Fortran は、デフォルトの kind 型付きパラメーターを 4 に設定します。 kind 型付きパラメーターは、整数値のバイト・サイズと同じです。 **-qintsize** コンパイラー・オプションを使用して、デフォルトの整数サイズを 2、4、8 バイトのいずれかに変更してください。 **-qintsize** オプションが、デフォルトの論理サイズに対しても同じように影

響することに注意してください。

IBM 拡張 の終り

整数のタイプ指定子は、**INTEGER** キーワードを含んでいなければなりません。整数のタイプ指定子のエンティティーの宣言に関する詳細については、 399 ページの『INTEGER』を参照してください。

符号付き整数のリテラル定数の形式は、以下のとおりです。



kind_param　　数字ストリング またはスカラー整数定数名 のいずれかです。

符号付き整数のリテラル定数は、オプションで符号を持ち、その後に小数点の付かない整数を表す 10 進数のストリングが続き、さらに必要な場合には kind 型付きパラメーターが続きます。符号付き整数のリテラル定数は、正、負、ゼロのいずれでもかまいません。無符号でゼロ以外の場合、その定数は正の値と見なされます。

kind_param を指定する場合、リテラル定数の大きさはその *kind_param* で許される値の範囲内で表現可能なものでなければなりません。

IBM 拡張

XL Fortran では、*kind_param* が指定されておらず、しかも定数の大きさをデフォルトの整数では表せない場合には、その定数は表現可能な形にプロモートされます。

XL Fortran は、内部的に 2 の補数表記で整数を表します。最左端のビットは数の符号です。

IBM 拡張 の終り

整定数の例

```
0                       ! has default integer size
-173_2                  ! 2-byte constant
9223372036854775807     ! Kind type parameter is promoted to 8
```

実数

IBM 拡張

次の表では、XL Fortran が実数データ型で表すことのできる値の範囲を示しています。

Kind パラメーター	ゼロ以外の近似絶対最小値	近似絶対最大値	近似精度 (10 進数)
4	1.175494E-38	3.402823E+38	7
8	2.225074D-308	1.797693D+308	15
16	2.225074Q-308	1.797693Q+308	31

XL Fortran は、デフォルトの kind 型付きパラメーターを 4 に設定します。 kind 型付きパラメーターは、実数値のバイト・サイズと等しい値を持ちます。 **-qrealsize** コンパイラー・オプションを使用して、デフォルトの実サイズを 4 バイトまたは 8 バイトに変更してください。 **-qrealsize** オプションはデフォルトの複素数のサイズに影響することに注意してください。

XL Fortran は、**REAL(4)** および **REAL(8)** の数を内部的には ANSI/IEEE の 2 進浮動小数点形式で表します。この形式は、符号ビット (s)、偏向指数 (e)、および小数部 (f) から構成されています。 **REAL(16)** 表示は、**REAL(8)** 形式に基づいています。

```
REAL(4)
ビット番号 0....|....1....|....2....|....3.
           seeeeeeeefffffffffffffffffffffff

REAL(8)
ビット番号 0....|....1....|....2....|....3....|....4....|....5....|....6...
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff

REAL(16)
ビット番号 0....|....1....|....2....|....3....|....4....|....5....|....6...
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
ビット番号 |....7....|....8....|....9....|....0....|....1....|....2....|..
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
```

また、この ANSI/IEEE 2 進浮動小数点形式は、+無限大、-無限大、非数値 (NaN)(not-a-number) という値も表します。 NaN はさらに、静止 NaN (NaNQ) と信号 NaN (NaNS) に分類することができます。 NaN 値の内部表示に関する詳細については、「ユーザーズ・ガイド」の『*XL Fortran 浮動小数点処理*』を参照してください。

IBM 拡張 の終り

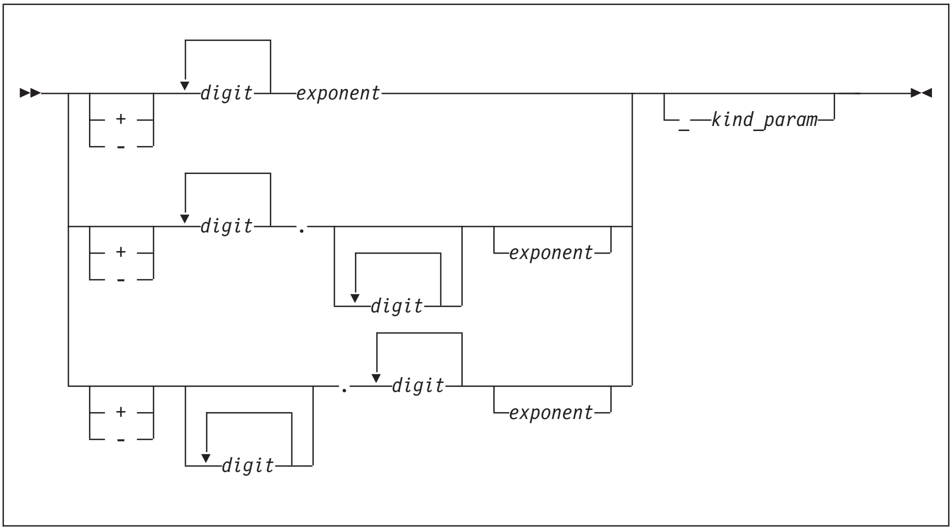
実数型指定子には、**REAL** キーワードか **DOUBLE PRECISION** キーワードが含まれていなければなりません。 **DOUBLE PRECISION** 値の精度はデフォルトの実数値の 2 倍です。 (*単精度* という用語は、IEEE の 4 バイト表記を、*倍精度* という用語は、

IEEE の 8 バイト表記を意味します。) 実数タイプのエンティティーの宣言に関する詳細については、 453 ページの『REAL』および 342 ページの『DOUBLE PRECISION』を参照してください。

実リテラル定数の形式は、次のとおりです。
- 基本実定数の後には、オプションとして、kind 型付きパラメーターが続きます。
- 基本実定数の後には、指数と、オプションとして kind 型付きパラメーターが続きます。
- 整定数 (*kind_param* を指定しない) の後には指数と、オプションとして kind 型付きパラメーターが続きます。

基本実定数は、順に、オプションの符号、整数部、小数点、および小数部から構成されています。整数部と小数部はどちらも数字ストリングです。これらのいずれか一方を省略することはできますが、両方を同時に省略することはできません。ユーザーは、定数の近似値を出すために XL Fortran が使用する桁数よりも多い桁数を使って基本実定数を書くことができます。 XL Fortran では、基本実定数は 10 進数として解釈されます。

実定数の形式は、次のとおりです。



exponent

kind_param 数字ストリング またはスカラー整数定数名 のいずれかです。

digit_string は、10 の累乗を示します。 **E** は、デフォルトの実定数のタイプを指定します。 **D** は、デフォルトの **DOUBLE PRECISION** タイプの定数を指定します。

IBM * XL Fortran では、**Q** は **REAL(16)** タイプの定数を指定します。 IBM

exponent と *kind_param* が共に指定された場合は、指数文字は **E** でなければなりません。 **D** または **Q** を指定した場合は、*kind_param* を指定することはできません。

指数および kind 型付きパラメーターが指定されない実リテラル定数は、デフォルトの実定数タイプです。

実定数の例

例 1:

```
+0.
```

例 2:

```
+5.432E02_16   ! 543.2 in 16-byte representation
```

例 3:

```
7.E3
```

IBM 拡張

例 4:

```
3.4Q-301       ! Extended-precision constant
```

IBM 拡張 の終り

複素数

複素数タイプの指定子には、次のいずれかが含まれていなければなりません。
- **COMPLEX** キーワード
- IBM XL Fortran では、**DOUBLE COMPLEX** キーワード IBM

複素数タイプのエンティティーの宣言に関する詳細については、 319 ページの『COMPLEX』および 339 ページの『DOUBLE COMPLEX』を参照してください。

IBM 拡張

次の表では、複素数タイプの指定子に **COMPLEX** キーワードが指定されている場合に、 XL Fortran が kind 型付きパラメーターおよび長さ指定について表すことのできる値を示しています。

Kind タイプ・パラメーター COMPLEX(*i*)	長さ指定 COMPLEX**j*
4	8
8	16
16	32

IBM 拡張 の終り

すべての FORTRAN コンパイラーにおいて、複素定数の種類は、実数部分と虚数部分の定数の種類によって決まります。

IBM 拡張

XL Fortran では、kind 型付きパラメーターにより複素数エンティティーの各部分の精度が指定され、長さ指定により複素数エンティティーの全体の長さが指定されます。

DOUBLE COMPLEX の値の精度は、デフォルトの複素数の 2 倍となります。

複素数タイプのスカラー値は、複素数コンストラクターを使って構成することができます。複素数コンストラクターの形式は次のとおりです。

```
►►—(——expression——,——expression——)——————————————————————————►◄
```

複素数のリテラル定数は、それぞれの式が一対の初期化式となっている複素数コンストラクターです。 複素数コンストラクターの各部分で、変数と式を XL Fortran 拡張機能として使用できます。

IBM 拡張 の終り

Fortran 95

Fortran 95 では、複素数コンストラクターの各部分に使用できるのは、単精度符号付き整数または実リテラル定数だけです。

Fortran 95 の終り

リテラル定数の両方の部分とも実数タイプである場合、リテラル定数の kind 型付きパラメーターは精度の高い方の kind パラメーターを使用し、精度の低い方の kind 型付きパラメーターは高い方の精度に変換されます。

両方の部分とも整数タイプである場合、それらはデフォルトの実数タイプに変換されます。一方が整数タイプで、もう一方が実数タイプである場合、整数タイプの方が、実数タイプの精度に変換されます。

複素数タイプのエンティティーの宣言に関する詳細については、 319 ページの『COMPLEX』および 339 ページの『DOUBLE COMPLEX』を参照してください。

IBM 拡張

複素数の各部分は、内部的には、次のように表現されます (符号ビット (s)、偏向指数 (e)、および小数部 (f))。

```
COMPLEX(4) (COMPLEX*8 に等しい)
ビット番号 0....|....1....|....2....|....3....|....4....|....5....|....6...
           seeeeeeeefffffffffffffffffffffffseeeeeeeefffffffffffffffffffffff

COMPLEX(8) (COMPLEX*16 に等しい)
ビット番号 0....|....1....|....2....|....3....|....4....|....5....|....6...
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
ビット番号 .|....7....|....8....|....9....|....0....|....1....|....2....|..
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff

COMPLEX(16) (COMPLEX*32 に等しい)
ビット番号 0....|....1....|....2....|....3....|....4....|....5....|....6...
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
ビット番号 .|....7....|....8....|....9....|....0....|....1....|....2....|..
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
ビット番号 ...3....|....4....|....5....|....6....|....7....|....8....|....9
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
ビット番号 ....|....0....|....1....|....2....|....3....|....4....|....5....
           seeeeeeeeeeeffffffffffffffffffffffffffffffffffffffffffffffffffff
```

IBM 拡張 の終り

複素定数の例

例 1:

```
(3_2,-1.86)        ! Integer constant 3 is converted to default real
                   ! for constant 3.0
```

IBM 拡張

例 2:

```
(45Q6,6D45)        ! The imaginary part is converted to extended
                   ! precision 6.Q45
```

例 3:

```
(1+1,2+2)          ! Use of constant expressions. Both parts are
                   ! converted to default real
```

IBM 拡張 の終り

論理

IBM 拡張

次の表は、 XL Fortran が論理データ型を使用して表すことのできる値を示しています。

Kind パラメーター	値	内部 (16 進) 表現
1	·TRUE. ·FALSE.	01 00
2	·TRUE. ·FALSE.	0001 0000
4	·TRUE. ·FALSE.	00000001 00000000
8	·TRUE. ·FALSE.	0000000000000001 0000000000000000

注: .TRUE. に対する 1、および .FALSE. に対する 0 以外の内部表現は未定義です。

XL Fortran は、デフォルトの kind 型付きパラメーターを 4 に設定します。 kind 型付きパラメーターは、論理値のバイト・サイズと同じになります。 **-qintsize** コンパイラー・オプションを使用して、デフォルトの論理サイズを 2、4、または 8 バイトのいずれかに変更してください。 **-qintsize** オプションがデフォルトの整数サイズに対しても同じように影響することに注意してください。

IBM 拡張 の終り

論理タイプの指定子は、**LOGICAL** キーワードを含んでいなければなりません。論理タイプのエンティティーの宣言に関する詳細については、 411 ページの『LOGICAL』を参照してください。

論理タイプのリテラル定数の形式は、次のとおりです。

```
>>-+-.TRUE.--+--+--------------+-------------------------------><
   '-.FALSE.-'  '-_--kind_param-'
```

kind_param
: *数字ストリング* または*スカラー整数定数名* のいずれかです。

論理定数は、真または偽のいずれかの論理値をとることができます。

IBM 拡張

.TRUE. および .FALSE. の代わりに、それぞれ省略形の T および F (ピリオドなし) を使用することもできます。ただし、この省略形は、定様式入力で使用する場合、あるいは **DATA** ステートメント、**STATIC** ステートメント、またはタイプ宣言ステートメントの初期値として使用する場合に限られます。 kind 型付きパラメーターについては、省略形を指定することはできません。 T または F が名前付き定数として定義されている場合、それは論理リテラル定数ではなく名前付き定数として処理されます。

IBM 拡張 の終り

論理定数の例

```
.FALSE._4
.TRUE.
```

文字

文字タイプの指定子は、**CHARACTER** キーワードを含んでいなければなりません。文字タイプの宣言に関する詳細については、 306 ページの『CHARACTER』を参照してください。

文字タイプのリテラル定数の形式は、次のとおりです。

```
>>-+------------------+--+-'--character_string--'-+-----------><
   '-kind_param--_----'  '-"--character_string--"-'
```

kind_param
    数字ストリング またはスカラー整数定数名 のいずれかです。

IBM 拡張

XL Fortran では ASCII の照合順序を表すために kind 型付きパラメーター値の 1 をサポートしています。

IBM 拡張 の終り

文字タイプのリテラル定数は、二重引用符またはアポストロフィのいずれかによって区切ることができます。

character_string を構成する文字は、XL Fortran で表現可能ならどのような文字でもかまいません。ただし、改行文字 (\n) は、ソース行の終わりを表す文字として解釈される

ので、使用しないでください。区切り文字となるアポストロフィ (') や二重引用符 (") は、その文字定数によって表されるデータの一部ではありません。これらの区切り文字の間に組み込まれたブランクは意味を持ちます。

ストリングをアポストロフィによって区切る場合、ストリング中でアポストロフィを表すには、間にブランクを入れずにアポストロフィを 2 つ続けて使用します。ストリングを二重引用符で区切る場合、ストリング中で二重引用符を表すには、間にブランクを入れずに二重引用符を 2 つ続けて使用します。 2 つの連続するアポストロフィまたは二重引用符は、1 文字として扱われます。

アポストロフィで囲んだ文字タイプのリテラル定数内に二重引用符を入れて、二重引用符を表すことができます。また、二重引用符で囲んだ文字定数内にアポストロフィを入れて、 1 つのアポストロフィを表すこともできます。

文字タイプのリテラル定数の長さは、区切り文字の間の文字数です。ただし連続する一対のアポストロフィや二重引用符は、1 文字としてカウントされます。

長さがゼロの文字オブジェクトでは、ストレージを使用しません。

IBM 拡張

XL Fortran では、それぞれの文字オブジェクトが 1 バイトのストレージを必要とします。

C 言語での使用法との整合性を確保するために、 XL Fortran では、文字ストリング内で次のエスケープ・シーケンスを認識します。

エスケープ	意味
\b	バックスペース
\f	用紙送り
\n	改行
\t	タブ
\0	ヌル
\'	アポストロフィ (ストリングは終了しません)
\"	二重引用符 (ストリングは終了しません)
\\	バックスラッシュ
\x	x。ここで x は任意の文字

C コンパイラーとの互換性確保のためにプロシージャー参照内のスカラー文字初期化式がヌル文字 (\0) で終わるようにするためには、**-qnullterm** コンパイラー・オプション

を使用してください (詳しい説明および例外については、「ユーザーズ・ガイド」の『**-qnullterm** オプション』を参照してください)。

すべてのエスケープ・シーケンスは 1 つの文字を表します。

IBM 拡張 の終り

これらのエスケープ・シーケンスを単一の文字として扱いたくない場合は、**-qnoescape** コンパイラー・オプションを指定してください。 (「ユーザーズ・ガイド」の『**-qescape** オプション』を参照してください。) バックスラッシュは、特別な意味を持たなくなります。

文字タイプのリテラル定数の最大長は、ステートメントに使用できる文字の最大数によって決まります。

IBM 拡張

-qctyplss コンパイラー・オプションを指定すると、文字定数は、ホレリス定数と同様に扱われます。ホレリス定数については、 65 ページの『ホレリス定数』を参照してください。 **-qctyplss** コンパイラー・オプションの内容については、「ユーザーズ・ガイド」の『**-qctyplss** オプション』を参照してください。

XL Fortran は、**-qmbcs** のコンパイラー・オプションによって、文字タイプのリテラル定数、ホレリス定数、 **H** 編集記述子、および注釈の中のマルチバイト文字をサポートします。

Unicode の文字およびファイル名もサポートします。環境変数 **LANG** が **UNIVERSAL** に設定され、**-qmbcs** コンパイラー・オプションが指定されている場合、コンパイラーは Unicode の文字およびファイル名の読み取りや書き込みを行うことができます。 (詳細については、「ユーザーズ・ガイド」を参照してください。)

IBM 拡張 の終り

文字定数の例

例 1:

```
''                   ! Zero-length character constant
```

例 2:

```
1_"ABCDEFGHIJ"       ! Character constant of length 10, with kind 1
```

IBM 拡張

例 3:

```
'\"\2\'\A567\\\\''   ! Character constant of length 10 "2'A567\\'
```

IBM 拡張 の終り

文字サブストリング

文字サブストリングとは、文字ストリング (親ストリングと呼ばれる) の連続した一部分で、スカラー変数名、スカラー定数、スカラー構造体コンポーネント、または配列エレメントのことです。文字サブストリングは、次の形式を持つサブストリング参照によって識別されます。

```
>>--+-scalar_variable_name-+--(--+-----------+--:--+-----------+--)------------><
    +-array_element--------+      '-int_expr1-'      '-int_expr2-'
    +-scalar_constant------+
    '-scalar_struct_comp---'
```

int_expr1 と *int_expr2*
: それぞれサブストリングの最左端および最右端の文字位置を示します。どちらの式もスカラー整数式で、サブストリング式と呼ばれます。

文字サブストリングの長さは、 `MAX(`*int_expr2* `-` *int_expr1* `+ 1,0)` の計算の結果です。

int_expr1 が *int_expr2* 以下の場合、その値は次の条件を満たしていなければなりません。

規則 1: 1 ≤ *int_expr1* ≤ *int_expr2* ≤ *length*

length は親ストリングの長さです。 *int_expr1* が省略されると、デフォルト値は 1 になります。 *int_expr2* が省略されると、デフォルト値は *length* になります。

IBM 拡張

XL Fortran の以前のバージョンは、上記の規則 1 で記したように、FORTRAN 77 の制約に準拠します。 FORTRAN 77 の規則に従ってサブストリング結合に関するコンパイル時チェックを実行するには、**-qnozerosize** コンパイラー・オプションを使用してください。Fortran 90 に対する準拠をチェックする場合は、**-qzerosize** を使用してください。サブストリング結合について実行時のチェックを行うには、**-qcheck** オプションと **-qzerosize** (または **-qnozerosize**) オプションの両方を使用します。 (詳細については、「ユーザーズ・ガイド」を参照してください。)

IBM 拡張 の終り

配列セクションのサブストリングの扱いはこれとは異なります。 100 ページの『配列セクションおよびサブストリングの範囲』を参照してください。

文字サブストリングの例:

```
CHARACTER(8) ABC, X, Y, Z
ABC = 'ABCDEFGHIJKL'(1:8)   ! Substring of a constant

X = ABC(3:5)                ! X = 'CDE'
Y = ABC(-1:6)               ! Not allowed in either FORTRAN 77 or Fortran 90

Z = ABC(6:-1)               ! Z = ' valid only in Fortran 90
```

バイト

IBM 拡張

XL Fortran では、バイト・タイプの指定子は **BYTE** キーワードです。バイト・タイプのエンティティーの宣言に関する詳細については、 299 ページの『BYTE』を参照してください。

BYTE 組み込みデータ型は、それ自身のリテラル定数形式を持っていません。 **BYTE** データ・オブジェクトは、その使用方法によって、**INTEGER(1)**、**LOGICAL(1)**、**CHARACTER(1)** のいずれかのデータ・オブジェクトとして扱われます。 66 ページの『タイプなし定数の使用方法』を参照してください。

IBM 拡張 の終り

派生型

派生型として知られる追加のデータ型は、組み込みデータ型や他の派生型から作成することができます。派生型の名前 (*type_name*)、データ型、および派生型のコンポーネントの名前を定義するには、タイプ定義が必要です。

IBM 拡張

レコード構造は、集合非配列データの操作でよく使用される拡張機能です。 Fortran 90 では、レコード構造は派生型よりも前に導入されました。

レコード構造で使用される構文は、多くの場合 Fortran 派生型で使用される構文に似ています。また、多くの場合、これらの 2 つの機能のセマンティクスも類似しています。このため、XL Fortran では、これらの 2 つの機能をほぼ完全に交換できるような方法でレコード構造がサポートされています。そのため、以下のようになります。

- どちらかの構文を使用して宣言された派生型のエンティティーは、 **TYPE** ステートメントまたは **RECORD** ステートメントのどちらかを使用して宣言できます。

- 派生型のオブジェクトのコンポーネントは、パーセント記号またはピリオドのどちらかを使用して選択できます。
- **レコード構造**宣言を使用して宣言された派生型は、構造体コンストラクターを持っています。
- 派生型のコンポーネントは、標準「等号」形式の初期化または拡張「ダブルスラッシュ」形式の初期化を使用して初期化できます。

ただし、以下の違いがあります。

- 標準派生型宣言では **%FILL** コンポーネントを使用できません。
- **レコード構造**宣言では **SEQUENCE** または **PRIVATE** ステートメントを使用してはいけません。
- **-qalign** オプションは、**レコード構造**宣言を使用して宣言された派生型にのみ適用されます。詳細については、「ユーザーズ・ガイド」に記載されている **-qalign** オプションを参照してください。
- **レコード構造**宣言を使用して宣言された派生型には、組み込みタイプと同じ名前を使用できます。
- **レコード構造**宣言を使用して宣言された派生型と標準の派生型宣言を使用して宣言された派生型とを判別するための規則には違いがあります。

IBM 拡張 の終り

```
>>--DERIVED_TYPE_statement--------------------------------------><

>>--+--------------------------+--------------------------------><
    '--PRIVATE_SEQUENCE_block--'

>>--component_def_stmt_block------------------------------------><

>>--END_TYPE_statement------------------------------------------><
```

DERIVED_TYPE_statement

構文の詳細については、 333 ページの『派生型 (TYPE)』を参照してください。

PRIVATE_SEQUENCE_block

PRIVATE ステートメント (キーワードのみ)、 **SEQUENCE** ステートメントのいずれかまたはその両方を含みます。それぞれのステートメントの 1 つだけを指定できます。構文の詳細については、 438 ページの『PRIVATE』および 467 ページの『SEQUENCE』を参照してください。

component_def_stmt_block

1 つ以上のタイプ宣言ステートメントから構成され、派生型のコンポーネントを定義します。タイプ宣言ステートメントには、**DIMENSION**、**POINTER**、および **ALLOCATABLE** 属性のみを指定できます。詳細な構文および説明に関しては、 481 ページの『タイプ宣言』を参照してください。

Fortran 95

さらに、Fortran 95 では、派生型の定義の各コンポーネントに対して、デフォルトの初期化を指定することができます。詳細な構文および説明に関しては、 481 ページの『タイプ宣言』を参照してください。

Fortran 95 の終り

END_TYPE_statement

356 ページの『END TYPE』を参照してください。

Fortran 95

Fortran 95 では派生型の直接コンポーネントには、次のものがあります。

- そのタイプのコンポーネント
- **ALLOCATABLE** または **POINTER** 属性が指定されていない派生型コンポーネントの直接コンポーネント。

Fortran 95 の終り

各派生型は、組み込みデータ型の最終コンポーネントとして解決されるか、あるいは割り当て可能またはポインターになります。

このタイプ名は、ローカル・エンティティーです。これは、どの組み込みデータ型とも同じ名前にすることはできません。ただし、**BYTE** および **DOUBLE COMPLEX** は例外です。

END TYPE ステートメントには、**TYPE** ステートメントで指定したものと同じ *type_name* をオプションで指定することができます。

派生型のコンポーネントでは、どの組み込みデータ型でも指定することができます。また、コンポーネントは、事前に定義された派生型の 1 つであってもかまいません。ポインター・コンポーネントは、そのコンポーネントの同じ派生データ型であってもかまいません。コンポーネントの名前は、派生型内では固有でなければなりませんが、派生型定義の有効範囲外の名前と異なっていてもかまいません。 **CHARACTER** タイプとして宣言されたコンポーネントは、定数宣言式である長さ指定を含んでいる必要があります。アスタリスクは、長さ指定子として使用できません。ポインター以外の配列コンポ

ーネントは、定数次元宣言子で宣言しなければなりません。ポインター配列コンポーネントは、*deferred_shape_spec_list* で宣言しなければなりません。

デフォルトでは、記憶順序は、コンポーネントの定義順序に暗黙指定されません。ただし、**SEQUENCE** ステートメントを指定すると、派生型は、*順序派生型* になります。順序派生型では、コンポーネントの順序に従って、この派生型で宣言されたオブジェクトの記憶順序が指定されます。順序派生型のコンポーネントが派生型の場合、その派生型も順序派生型でなければなりません。

順序派生型のサイズは、その派生型のすべてのコンポーネントを保持するために必要なストレージのバイト数に等しくなります。

順序派生型を使用すると、データの並びが揃わなくなることがあります。これは、プログラムのパフォーマンスに悪影響を与えます。

PRIVATE ステートメントは、派生型がモジュールの指定部分内で定義された場合にのみ、指定することができます。派生型のコンポーネントがプライベートと宣言されるタイプの場合、派生型の定義で **PRIVATE** ステートメントを指定するか、または派生型自体がプライベートでなければなりません。

タイプ定義がプライベートである場合、次のものは、定義するモジュール内でのみアクセス可能となります。
- タイプ名
- タイプの構造体コンストラクター
- タイプのエンティティー
- 仮引き数またはタイプの関数結果を持つプロシージャー

派生型の定義に、**PRIVATE** ステートメントが含まれている場合、その派生型が public であっても、そのコンポーネントは、定義するモジュール内でのみアクセスできます。構造体コンポーネントは、定義するモジュール内でのみ使用できます。

派生型オブジェクトのコンポーネントは、オブジェクトのいずれかの最終コンポーネントが I/O ステートメントの有効範囲の単位でアクセスできない場合、 I/O リスト項目として指定することはできません。派生型オブジェクトは、ポインターまたは割り当て可能であるコンポーネントを持っている場合、データ転送ステートメント内に指定することはできません。

派生型のスカラー・エンティティーは、*構造体* と呼ばれます。順序派生型のスカラー・エンティティーは、*順序構造体* と呼ばれます。構造体のタイプ指定子は、**TYPE** キーワードを含み、その後に括弧で囲まれた派生型の名前が続きます。指定する派生型のエンティティーの宣言に関する詳細は、 476ページのタイプ定義ステートメント『**TYPE**』を参照してください。構造体のコンポーネントは、*構造体コンポーネント* と呼ばれます。*構造体コンポーネント* は、構造体の 1 つのコンポーネントであるか、派生型の配列を成しているコンポーネントです。

private 派生型のオブジェクトは、定義するモジュールの外部で使用することはできません。

デフォルトの初期化は、等号の後ろに初期化式を付けるか、またはスラッシュで囲まれた *initial_value_list* を使用して指定できます。現在、この形式の初期化は、**レコード構造**宣言または標準の派生型宣言を使用して宣言されたコンポーネントでサポートされています。

Fortran 95

Fortran 95 で、デフォルトの初期化の候補となるデータ・オブジェクトは、以下のような名前付きデータ・オブジェクトです。

1. その直接コンポーネントのいずれにもデフォルトの初期化が指定されている派生型のデータ・オブジェクト
2. **POINTER** 属性と **ALLOCATABLE** 属性のどちらもないデータ・オブジェクト
3. 使用関連付けまたはホスト関連付けでないデータ・オブジェクト
4. ポイント先でないデータ・オブジェクト

非ポインター・コンポーネントのデフォルトの初期化は、そのタイプの直接コンポーネントに対して行われるどのデフォルト初期化よりも優先します。

仮引き数に **INTENT(OUT)** が指定されているものが、デフォルトの初期化がされている派生型である場合、それを想定サイズ配列にすることはできません。非ポインター・オブジェクトまたはサブオブジェクトがタイプ定義内でデフォルトの初期化を指定されている場合、それを **DATA** ステートメントによって初期化することはできません。

Fortran 95 の終り

デフォルトの初期化付きの派生型のデータ・オブジェクトは、 IBM 拡張として共通ブロックの中に指定できます。また、XL Fortran ではデフォルトの初期化は、**-qsave=defaultinit** が指定されていない限り、**SAVE** 属性を暗黙指定しません。

Fortran 95

明示的な初期化とは異なり、コンポーネントのデフォルトの初期化を有効にするために、データ・オブジェクトに **SAVE** 属性を指定する必要はありません。デフォルトの初期化は、派生型のいくつかのコンポーネントに対して指定することができますが、すべてのコンポーネントに指定する必要はありません。

ストレージに関連した記憶単位のデフォルトの初期化を指定することができます。ただし、デフォルトの初期化を提供するデフォルトのオブジェクトまたはサブオブジェクト

は、同じタイプでなければなりません。そのオブジェクトまたはサブオブジェクトは、同じ型付きパラメーターを持たなければならず、記憶単位に同じ値を提供しなければなりません。

直接コンポーネントは、タイプ定義内で対応するコンポーネント定義に対してデフォルトの初期化を指定していれば、コンポーネントへのアクセス可能性に関係なく、初期値を受け取ることになります。

デフォルトの初期化の対象になるデータ・オブジェクトの場合、その非ポインター・コンポーネントは、最初に定義されているか、または対応するデフォルトの初期化式によって定義されます。そのポインター・コンポーネントは、最初に関連解除されているか、または以下の条件のいずれかが満たされる場合は、関連解除されます。

- 最初に定義または関連解除されている場合
  - 当該のデータ・オブジェクトが **SAVE** 属性を持っている。
  - 当該のデータ・オブジェクトが、**BLOCK DATA** 単位、モジュール、またはメインプログラム単位で宣言されている。
- 定義または関連解除される場合
  - 関数が、その結果としての当該のデータ・オブジェクトとともに呼び出される。
  - プロシージャーが、**INTENT(OUT)** 仮引き数としての当該のデータ・オブジェクトとともに呼び出される。
  - プロシージャーが、ローカル・オブジェクトとしての当該のデータ・オブジェクトとともに呼び出され、そのデータ・オブジェクトに **SAVE** 属性がない。

コンポーネントに対するデフォルトの初期化を指定した派生型のオブジェクトの割り振りによって、コンポーネントは以下のようになります。

- 非ポインター・コンポーネントの場合、定義される。
- ポインター・コンポーネントの場合、関連解除される。

ENTRY ステートメントを持つサブプログラムでは、デフォルトの初期化は、参照されるプロシージャー名の引き数リストにある仮引き数の場合だけ実行されます。この種の仮引き数に **OPTIONAL** 属性がある場合、デフォルトの初期化はこの仮引き数がある場合に限り実行されます。

デフォルトの初期化が指定された派生型のモジュール・データ・オブジェクトが、デフォルト初期化の候補のデータ・オブジェクトである場合には、 **SAVE** 属性がなければなりません。

Fortran 95 の終り

標準の派生型宣言を使用して宣言された順次派生型のサイズは、その派生型のすべてのコンポーネントを保持するために必要なバイト数の合計と等しくなります。

レコード構造宣言を使用して宣言された順序派生型のサイズは、その派生型のコンポーネントおよび埋め込みのすべてを保持するために必要なバイト数の合計と等しくなります。

以前は、共通ブロックにある数値順序または文字順序の構造体は、そのコンポーネントが共通ブロック内で直接列挙されているかのように処理されていました。現在は、これは、標準の派生型宣言を使用して宣言されたタイプの構造体にのみ適用されます。

入出力

名前リスト入力では、構造はその非充てん最終コンポーネントのリストに展開されます。

名前リスト出力では、構造はその非充てん最終コンポーネントの値に展開されます。

定様式データ転送ステートメント (**READ**、**WRITE**、または **PRINT**) では、**%FILL** コンポーネントではない派生型のエンティティーのコンポーネントのみが *input-item-list* または *output-item-list* に指定されているかのように処理されます。

派生型のエンティティー内の **%FILL** フィールドは、定様式データ転送ステートメントでは埋め込みとして扱われます。

派生型のタイプの決め方

2 つのデータ・オブジェクトを、同じ派生型定義を参照することによって宣言している場合には、これらのデータ・オブジェクトの派生型は同じです。

これらのデータ・オブジェクトが別々の有効範囲単位内にある場合は、両者が同じ派生型を持つことができます。次の条件が満たされる場合は、ホスト関連付けまたは使用関連付けを介して派生型定義にアクセスできるか、またはデータ・オブジェクトが自己の派生型定義を参照します。

- 派生型定義が、標準の派生型宣言を使用して宣言され、名前が同じであり、**SEQUENCE** プロパティーを持っており、しかも **PRIVATE** アクセス可能性を持たず、かつ順序、名前、属性が一致するコンポーネントを持っている。または、
- 派生型定義が、名前なしではないレコード構造宣言を使用して宣言され、名前が同じであり、**%FILL** コンポーネントを持っておらず、かつ順序と属性が一致するコンポーネントを持っており、しかも **%FILL** コンポーネントが 2 つのデータ・オブジェクトの同じ位置に指定されている。

SEQUENCE を指定する派生型定義は、プライベートと宣言される定義、あるいはプライベートであるコンポーネントを持つ定義とは異なります。

派生型でのタイプ決定の例

```
PROGRAM MYPROG

TYPE NAME                           ! Sequence derived type
```

```
   SEQUENCE
   CHARACTER(20) LASTNAME
   CHARACTER(10) FIRSTNAME
   CHARACTER(1)  INITIAL
END TYPE NAME
TYPE (NAME) PER1

CALL MYSUB(PER1)
PER1 = NAME('Smith','John','K')   ! Structure constructor
CALL MYPRINT(PER1)

CONTAINS
  SUBROUTINE MYSUB(STUDENT)       ! Internal subroutine MYSUB
     TYPE (NAME) STUDENT          ! NAME is accessible via host association
              ...
  END SUBROUTINE MYSUB
END

SUBROUTINE MYPRINT(NAMES)         ! External subroutine MYPRINT
  TYPE NAME                       ! Same type as data type in MYPROG
     SEQUENCE
     CHARACTER(20) LASTNAME
     CHARACTER(10) FIRSTNAME
     CHARACTER(1)  INITIAL
  END TYPE NAME
  TYPE (NAME) NAMES               ! NAMES and PER1 from MYPROG
  PRINT *, NAMES                  ! have the same data type
END SUBROUTINE
```

異なるコンポーネント名の例

```
MODULE MOD
  STRUCTURE /S/
   INTEGER I
   INTEGER, POINTER :: P
  END STRUCTURE
  RECORD /S/ R
END MODULE
PROGRAM P
  USE MOD, ONLY: R
  STRUCTURE /S/
    INTEGER J
    INTEGER, POINTER :: Q
  END STRUCTURE
  RECORD /S/ R2
  R = R2 ! OK - same type name, components have same attributes and
         ! type (but different names)
END PROGRAM P
```

構造体コンポーネント

構造体コンポーネントは、派生型も含めどの明示タイプでもかまいません。

注: 構造体コンポーネントが、配列または配列セクションであるサブオブジェクトを持つ場合、 95 ページの『配列セクション』からのバックグラウンド情報が必要となり

ます。これについては、 100 ページの『配列セクションおよび構造体コンポーネント』に記述しています。スカラー構造体コンポーネントに関する以下の規則は、配列サブオブジェクトを持つ構造体コンポーネントにも適用されます。

コンポーネント*指定子* を使って特定の構造体コンポーネントを参照することもできます。スカラー・コンポーネント指定子は次のような構文を持ちます。

scalar_struct_comp:

```
>>--name--+------------------------+--------------------------------->
          '-(--int_expr_list--)-'

>--+--------------------------------------------------------+------->
   | .----------------------------------------------------. |
   | V                                                    | |
   '---separator comp_name--+------------------------+-+--'
                            '-(--int_expr_list--)-'

>--separator comp_name--+------------------------+------------------><
                        '-(--int_expr_list--)-'
```

name	派生型のオブジェクトの名前です。
comp_name	派生型コンポーネントの名前です。
int_expr	添え字式と呼ばれるスカラー整数または実数式です。
separator	**%** または **.** です。 **注: .** (ピリオド) は IBM 拡張です。

構造体コンポーネントは、最右端の *comp_name* と同様のタイプ、型付きパラメーター、**POINTER** 属性 (もしあれば) を持ちます。構造体コンポーネントは、親オブジェクトから **INTENT**、**TARGET**、**PARAMETER** などの属性を継承します。

注:

1. 各 *comp_name* は、直前の *name*、*comp_name* のいずれかのコンポーネントでなければなりません。
2. *name* および最右端を除く各 *comp_name* は、派生型でなければなりません。
3. *int_expr_list* 内の添え字式の数は、先行する *name* または *comp_name* のランクと等しくなければなりません。
4. *name* またはいずれかの *comp_name* が、配列の名前である場合、*int_expr_list* を持たなければなりません。
5. 最右端の *comp_name* は、スカラーでなければなりません。

名前リストのフォーマット設定では、区切り文字はパーセント記号でなければなりません。

区切り文字としてピリオドを使用する構造体コンポーネントであるか、または 2 進演算であるかのいずれにでも解釈される形式が式に入っており、その名前の演算子が有効範囲単位内でアクセス可能な場合、 XL Fortran はその式を 2 進演算として処理します。それが意図した解釈ではない場合は、パーセント記号を使用してその部分への参照を解除するか、または、自由ソース形式であれば、ピリオドと *comp_name* の間に空白を挿入してください。

構造体コンポーネントへの参照の例:

例 1: 区切り文字としてのピリオドのあいまいな使用

```
MODULE MOD
  STRUCTURE /S1/
    STRUCTURE /S2/ BLUE
      INTEGER I
    END STRUCTURE
  END STRUCTURE
  INTERFACE OPERATOR(.BLUE.)
    MODULE PROCEDURE BLUE
  END INTERFACE
CONTAINS
  INTEGER FUNCTION BLUE(R1, I)
    RECORD /S1/ R1
    INTENT(IN) :: R1
    INTEGER, INTENT(IN) :: I
    BLUE = R1%BLUE%I + I
  END FUNCTION BLUE
END MODULE MOD

PROGRAM P
  USE MOD
  RECORD /S1/ R1
  R1%BLUE%I = 17
  I = 13
  PRINT *, R1.BLUE.I ! Calls BLUE(R1,I) - prints 30
  PRINT *, R1%BLUE%I ! Prints 17
END PROGRAM P
```

例 2: 区切り文字の混合

```
STRUCTURE /S1/
  INTEGER I
END STRUCTURE
STRUCTURE /S2/
  RECORD /S1/ C
END STRUCTURE
RECORD /S2/ R
R.C%I = 17 ! OK
```

```
R%C.I = 3 ! OK
R%C%.I = 13 ! OK
R.C.I = 19 ! OK
END
```

例 3: 派生型でのパーセントとピリオドの機能

```
STRUCTURE /S/
  INTEGER I, J
END STRUCTURE
TYPE DT
  INTEGER I, J
END TYPE DT
RECORD /S/ R1
TYPE(DT) :: R2
R1.I = 17; R1%J = 13
R2.I = 19; R2%J = 11
END
```

割り振り可能コンポーネント

IBM 拡張

ポインター・コンポーネントと同様、割り振り可能コンポーネントは最終コンポーネントとして定義されます。これは、値 (存在する場合) が残りの構造とは別に保管され、構造の作成時にはこのストレージが存在しない (オブジェクトが割り振り解除されているため) ためです。最終ポインター・コンポーネントと同様、最終割り振り可能コンポーネントを含む変数は入出力リストに直接入れることは禁じられています。入出力の割り振り可能コンポーネントまたはポインター・コンポーネントは、ユーザーがリストします。

現在、割り振り可能配列と同様に、割り振り可能コンポーネントはストレージ関連付けコンテキストを禁じられています。このため、最終割り振り可能コンポーネントを含む変数は、 **COMMON** または **EQUIVALENCE** には入れることができません。ただし、**SEQUENCE** タイプには割り振り可能コンポーネントが許可されており、同じタイプを複数の有効範囲単位に別個に定義することが許されます。

最終割り振り可能コンポーネントを含む変数を割り振り解除すると、現在割り振られている変数の割り振り可能コンポーネントがすべて自動的に割り振り解除されます。

割り振り可能コンポーネントを含む派生型の構造コンストラクターでは、割り振り可能コンポーネントに対応する式は以下のいずれかでなければなりません。

- 組み込み関数 NULL() への引き数なしの参照。割り振り可能コンポーネントは、現在割り振り済みでないという割り振り状況を受け取ります。

- それ自体割り振り可能な変数。割り振り可能コンポーネントは、変数の割り振り状況を受け取ります。また、変数がすでに割り振られている場合は、割り振り状況に加え、変数の値も受け取ります。変数が割り振り済み配列の場合、割り振り可能コンポーネントも変数の境界を持ちます。
- その他の任意の式。割り振り可能コンポーネントは、式と同じ値で現在割り振り済みであるという割り振り状況を受け取ります。式が配列の場合、割り振り可能コンポーネントは同じ境界を持ちます。

割り振り可能コンポーネントを含む派生型のオブジェクトの組み込み割り当ての場合、左側にある変数の割り振り可能コンポーネントは、割り振り状況と、割り振り済みである場合は式の対応するコンポーネントの境界と値を受け取ります。これは、次の順序でステップが実行されたかのように行われます。

1. 変数のコンポーネントが現在割り振られている場合、このコンポーネントが割り振り解除されます。
2. 式の対応するコンポーネントが現在割り振り済みである場合、変数のコンポーネントが同じ境界で割り振られます。これで、式のコンポーネントの値が、組み込み割り当てを使用して変数の対応するコンポーネントに割り当てられます。

INTENT(OUT) 仮引き数に関連した実引き数の割り振り済み最終割り振り可能コンポーネントはプロシージャーに入るときに割り振り解除されるので、仮引き数の対応するコンポーネントは、現在割り振り済みでないという割り振り状況を持つようになります。

これにより、変数の割り振り可能コンポーネントの前の内容を指し示すポインターが、確実に未定義になります。

例:

```
MODULE REAL_POLYNOMIAL_MODULE
  TYPE REAL_POLYNOMIAL
    REAL, ALLOCATABLE :: COEFF(:)
  END TYPE
  INTERFACE OPERATOR(+)
    MODULE PROCEDURE RP_ADD_RP, RP_ADD_R
  END INTERFACE
CONTAINS
  FUNCTION RP_ADD_R(P1,R)
    TYPE(REAL_POLYNOMIAL) RP_ADD_R, P1
    REAL R
    INTENT(IN) P1,R
    ALLOCATE(RP_ADD_R%COEFF(SIZE(P1%COEFF)))
    RP_ADD_R%COEFF = P1%COEFF
    RP_ADD_R%COEFF(1) = P1%COEFF(1) + R
  END FUNCTION
  FUNCTION RP_ADD_RP(P1,P2)
    TYPE(REAL_POLYNOMIAL) RP_ADD_RP, P1, P2
    INTENT(IN) P1, P2
    INTEGER M
    ALLOCATE(RP_ADD_RP%COEFF(MAX(SIZE(P1%COEFF), SIZE(P2%COEFF))))
```

```
    M = MIN(SIZE(P1%COEFF), SIZE(P2%COEFF))
    RP_ADD_RP%COEFF(:M) = P1%COEFF(:M) + P2%COEFF(:M)
    IF (SIZE(P1%COEFF)>M) THEN
      RP_ADD_RP%COEFF(M+1:) = P1%COEFF(M+1:)
    ELSE IF (SIZE(P2%COEFF)>M) THEN
      RP_ADD_RP%COEFF(M+1:) = P2%COEFF(M+1:)
    END IF
  END FUNCTION
END MODULE

PROGRAM EXAMPLE
  USE REAL_POLYNOMIAL_MODULE
  TYPE(REAL_POLYNOMIAL) P, Q, R
  P = REAL_POLYNOMIAL((/4,2,1/)) ! Set P to (X**2+2X+4)
  Q = REAL_POLYNOMIAL((/1,1/)) ! Set Q to (X+1)
  R = P + Q ! Polynomial addition
  PRINT *, 'Coefficients are: ', R%COEFF
END
```

IBM 拡張 の終り

構造体コンストラクター

```
►►—type_name—(—expr_list—)——————————————————————————————————►◄
```

type_name　派生型の名前です。

expr　式です。式は、 107 ページの『第 5 章 式および割り当て』で定義されます。

構造体コンストラクターによって、値の番号付きリストから派生型のスカラー値を構成することができます。構造体コンストラクターは、参照される派生型の定義の前に指定できません。

expr_list には、派生型のコンポーネントごとに 1 つの値が含まれています。 *expr_list* の式の順序は、数および順序の点で派生型のコンポーネントと一致していなければなりません。それぞれの式のタイプと型付きパラメーターは、対応するコンポーネントのタイプおよび型付きパラメーターと整合性のある割り当てになっている必要があります。データ型は、必要に応じて変換されます。

ポインターであるコンポーネントは、ポインターのコンポーネントと同じタイプで宣言できます。構造体コンストラクターがポインターを含む派生型に対して作成された場合、ポインター・コンポーネントに対応する式は、ポインター割り当てステートメント内のそのようなポインターのために使用可能なターゲットとなり得るオブジェクトに対

して評価を行わなければなりません。

IBM 拡張

派生型のコンポーネントが割り振り可能な場合、対応するコンストラクター式は、引き数なしの組み込み関数 **NULL()** への参照になるか、割り振り可能エンティティーになるか、あるいは対応するコンストラクター式の評価の結果が同じランクのエンティティーになります。式が組み込み関数 **NULL()** への参照の場合、コンストラクターの対応するコンポーネントは、現在割り振り済みでないという状況を持ちます。式が割り振り可能エンティティーの場合、コンストラクターの対応するコンポーネントは、割り振り可能エンティティーと同じ割り振り状況を持ち、 割り振り済みである場合は、同じ境界 (もしあれば) と値も持ちます。そうでない場合は、コンストラクターの対応するコンポーネントは、現在割り振り済みであるという割り振り状況と、式と同じ境界 (もしあれば) および値を持ちます。

レコード構造宣言を使用するコンポーネントが **%FILL** の場合、そのタイプの構造コンストラクターは使用できません。

派生型が有効範囲単位内でアクセス可能であり、有効範囲単位内でアクセス可能な同じ名前の派生型ではないクラス 1 のローカル・エンティティーがある場合、そのタイプの構造コンストラクターをその有効範囲内で使用することはできません。

IBM 拡張 の終り

派生型の例:

例 1:

```
MODULE PEOPLE
  TYPE NAME
     SEQUENCE                     ! Sequence derived type
     CHARACTER(20) LASTNAME
     CHARACTER(10) FIRSTNAME
     CHARACTER(1)  INITIAL
  END TYPE NAME

  TYPE PERSON                     ! Components accessible via use
                                  ! association
     INTEGER AGE
     INTEGER BIRTHDATE(3)         ! Array component
     TYPE (NAME) FULLNAME         ! Component of derived type
  END TYPE PERSON
END MODULE PEOPLE
PROGRAM TEST1
  USE PEOPLE
  TYPE (PERSON) SMITH, JONES
  SMITH = PERSON(30, (/6,30,63/), NAME('Smith','John','K'))
                                  ! Nested structure constructors
  JONES%AGE = SMITH%AGE           ! Component designator
```

```
  CALL TEST2
  CONTAINS

  SUBROUTINE TEST2
    TYPE T
      INTEGER EMP_NO
      CHARACTER, POINTER :: EMP_NAME(:)  ! Pointer component
    END TYPE T
    TYPE (T) EMP_REC
    CHARACTER, TARGET :: NAME(10)
    EMP_REC = T(24744,NAME)              ! Pointer assignment occurs
  END SUBROUTINE                         ! for EMP_REC%EMP_NAME
END PROGRAM
```

Fortran 95

例 2:

```
PROGRAM LOCAL_VAR
   TYPE DT
      INTEGER A
      INTEGER :: B = 80
   END TYPE

   TYPE(DT) DT_VAR                      ! DT_VAR%B IS INITIALIZED
END PROGRAM LOCAL_VAR
```

例 3:

```
MODULE MYMOD
   TYPE DT
      INTEGER :: A = 40
      INTEGER, POINTER :: B => NULL()
   END TYPE
END MODULE

PROGRAM DT_INIT
   USE MYMOD
   TYPE(DT), SAVE :: SAVED(8)            ! SAVED%A AND SAVED%B ARE INITIALIZED
   TYPE(DT) LOCAL(5)                     ! LOCAL%A LOCAL%B ARE INITIALIZED
END PROGRAM
```

Fortran 95 の終り

レコード構造

IBM 拡張

レコード構造の宣言

レコード構造を宣言すると、標準の FORTRAN 派生型定義でユーザー定義タイプを宣言するのと同じ方法でユーザー定義タイプが宣言されます。**レコード構造**宣言を使用し

て宣言されたタイプは派生型です。多くの場合、標準の FORTRAN 構文を使用して宣言された派生型に適用される規則は、レコード構造構文を使用して宣言された派生型に適用されます。規則に違いがある場合、**レコード構造**宣言を使用して宣言された派生型と、標準の派生型宣言を使用して宣言された派生型の両者を参照するとその違いがわかります。

レコード構造宣言は、以下の構文に従って行います。

record_structure_dcl:

```
>>--structure_stmt----------------------------------------------><

      .-----------------------.
      V                       |
>>------struct_comp_dcl_item--+---------------------------------><

>>--end_structure_stmt------------------------------------------><
```

struct_comp_dcl_item:

```
>>--+-component_def_stmt---+------------------------------------><
    +-record_structure_dcl-+
    '-parameter_stmt-------'
```

component_def_stmt は、派生型のコンポーネントを定義するために使用されるタイプ宣言ステートメントです。

structure_stmt:

```
>>--STRUCTURE--+------------------+--+--------------------+-----><
               '-/structure_name/-'  '-component_dcl_list-'
```

component_dcl:

```
>>--a--+---------------+----------------------------------------><
       '-(-array_spec-)'
```

a はオブジェクト名です。

構造ステートメントは、それを囲む最も近いプログラム単位、インターフェース本体、またはサブプログラムの有効範囲単位内の派生型として、*structure_name* を宣言します。派生型は、その有効範囲単位内のクラス 1 のローカル・エンティティーです。

構造ステートメントは別の**レコード構造**宣言内にネストされない限り、*component_dcl_list* は指定されません。同様に構造ステートメントの *structure_name*

は、別のレコード構造宣言内にネストされている *record_structure_dcl* の一部でない限り省略できません。 *record_structure_dcl* は、少なくとも 1 つのコンポーネントを持っている必要があります。

レコード構造宣言を使用して宣言された派生型は、順序派生型であり、順序派生型に適用されるすべての規則に従います。標準派生型宣言を使用して宣言された順序派生型と同様に、**レコード構造**宣言を使用して宣言されたタイプのコンポーネントは、非順序派生型にできません。**レコード構造**宣言には、**PRIVATE** または **SEQUENCE** ステートメントを入れることはできません。

レコード構造宣言は有効範囲単位を定義します。 *record_structure_dcl* にあるすべてのステートメントは、レコード構造宣言の有効範囲単位の中に入ります (*record_structure_dcl* に含まれているその他の *record_structure_dcl* は例外)。これらの規則は、標準の派生型宣言にも当てはまります。明確にするために、以下で繰り返して説明します。

record_structure_dcl 内の *parameter_stmt* は、それを囲む最も近いプログラム単位、インターフェース本体、またはサブプログラムの有効範囲単位内にある名前付き定数を宣言します。このような *parameter_stmt* で宣言された名前付き定数は、この定数を含む *record_structure_dcl* で宣言されたコンポーネントと同じ名前にできます。

structure_stmt で宣言されたコンポーネントはどれも、それを囲む派生型のコンポーネントであり、それを囲む構造の有効範囲単位のローカル・エンティティーです。このようなコンポーネントのタイプは、その *structure_stmt* で宣言される派生型です。

標準の派生型宣言を使用して宣言された派生型とは異なり、**レコード構造**宣言を使用して宣言された派生型名は、組み込みタイプの名前と同じ名前にできます。

レコード構造宣言の *component_def_stmt* には、コンポーネントの名前の代わりに **%FILL** を使用できます。 **%FILL** コンポーネントは、**レコード構造**宣言において、データを希望する位置に合わせるためのプレースホルダーとして使用されます。初期化では、**%FILL** コンポーネントに関する指定はできません。**レコード構造**宣言内の **%FILL** の各インスタンスは、タイプに指定したその他のすべてのコンポーネント名と異なり、しかもその他のすべての **%FILL** コンポーネントとも異なる、固有のコンポーネント名として扱われます。 **%FILL** は 1 つのキーワードであり、**-qmixed** コンパイラー・オプションの影響は受けません。

名前を持たない、ネストされた構造の各インスタンスは、他のすべてのアクセス可能エンティティーの名前とは異なる固有の名前を持っているものとして扱われます。

派生型についてこれまで説明した規則の延長として、**レコード構造**宣言を使用して宣言された派生型の直接コンポーネントには、次のものがあります。

- タイプが **%FILL** コンポーネントではないコンポーネント
- **POINTER** 属性を持っておらず、**%FILL** コンポーネントではない派生型コンポーネントの直接コンポーネント。

派生型の非充てん最終コンポーネントは派生型の最終コンポーネントであり、直接コンポーネントでもあります。

以前は、デフォルトの初期化が行われる派生型のオブジェクトを共通ブロックに入れることはできませんでした。現在は、これは拡張機能として可能になりました。ただし、共通ブロックの初期化が、複数の有効範囲単位内で行われることがないようにする必要があります。

レコード構造の宣言の例:

例 1: ネストされたレコード構造宣言 - 名前付きおよび名前なし

```
STRUCTURE /S1/
  STRUCTURE /S2/ A ! A is a component of S1 of type S2
    INTEGER I
  END STRUCTURE
  STRUCTURE B ! B is a component of S1 of unnamed type
    INTEGER J
  END STRUCTURE
END STRUCTURE
RECORD /S1/ R1
RECORD /S2/ R2 ! Type S2 is accessible here.
R2.I = 17
R1.A = R2
R1.B.J = 13
END
```

例 2: 構造宣言内でネストされているパラメーター・ステートメント

```
INTEGER I
STRUCTURE /S/
  INTEGER J
  PARAMETER(I=17, J=13) ! Declares I and J in scope of program unit to
                        ! be named constants
END STRUCTURE
INTEGER J ! Confirms implicit typing of named constant J
RECORD /S/ R
R.J = I + J
PRINT *, R.J ! Prints 30
END
```

例 3: %FILL フィールド

```
STRUCTURE /S/
  INTEGER I, %FILL, %FILL(2,2), J
  STRUCTURE /S2/ R1, %FILL, R2
    INTEGER I
  END STRUCTURE
END STRUCTURE
RECORD /S/ R
PRINT *, LOC(R%J)-LOC(R%I)  ! Prints 24 with -qintsize=4
PRINT *, LOC(R%R2)-LOC(R%R1) ! Prints 8 with -qintsize=4
END
```

ストレージ・マッピング

レコード構造宣言を使用して宣言された派生型は、順序派生型です。このようなタイプのオブジェクトのコンポーネントは指定された順序でメモリーに保管されます。標準の派生型宣言を使用して宣言された順序派生型のオブジェクトについてもこれと同じことが言えます。

-qalign オプションは、ストレージ内でのデータ・オブジェクトの位置合わせを指定します。これにより、誤って位置合わせされたデータによるパフォーマンス上の問題が回避されます。 **[no]4k** と **struct** の両方のサブオプションを指定でき、相互に排他的ではありません。デフォルトの設定は **-qalign=no4k:struct=natural** です。 [no]4K は、主に、論理ボリューム入出力とディスク・ストライピングの組み合わせで役に立ちます。

IBM 拡張 の終り

UNION および MAP

IBM 拡張

UNION は、囲まれた**レコード構造**の中に、プログラム内のデータ域を共有できるフィールド・グループを宣言します。

UNION および MAP は次の構文に従います。

union_dcl:

```
           ┌──────────────┐
           ▼              │
►►──UNION───union_dcl_item┴──────────────────────────────────────►◄

►►──END UNION────────────────────────────────────────────────────►◄
```

union_dcl_item:

```
►►──┬─map_dcl────────┬───────────────────────────────────────────►◄
    └─parameter_stmt─┘
```

map_dcl:

```
        ┌──────────────┐
        ▼              │
►►─MAP───map_dcl_item──┴─────────────────────────────────────►◄

►►─END MAP───────────────────────────────────────────────────►◄
```

map_dcl_item:

```
►►──┬─struct_comp_dcl_item─┬─────────────────────────────────►◄
    └─record_stmt──────────┘
```

struct_comp_dcl_item:

```
►►──┬─component_def_stmt───┬─────────────────────────────────►◄
    ├─record_structure_dcl─┤
    ├─parameter_stmt───────┤
    └─union_dcl────────────┘
```

UNION 宣言は**レコード構造**内に定義する必要があり、 **MAP** 宣言の中に入れることが可能です。また、**MAP** 宣言は **UNION** 宣言内になければなりません。 UNION 宣言内の *map_dcl_item* にあるすべての宣言は、宣言がどの *map_dcl* にあるのかにかかわらず同じネスト・レベルになっていなければなりません。このため、*map_dcl* 内のコンポーネント名を、同じレベルの他の *map_dcl* に入れることができません。

MAP 宣言内で宣言されるコンポーネントには **POINTER** または **ALLOCATABLE** 属性があってはいけません。

UNION MAP を持つレコード構造は I/O ステートメントには使用できません。

MAP 宣言で宣言されたコンポーネントは、**UNION** 構造体内のその他の MAP 宣言で宣言されたコンポーネントと同じストレージを共用します。ある **MAP** 宣言内にある 1 つのコンポーネントに値を割り当てると、ストレージをこのコンポーネントと共用する、その他の MAP 宣言内のコンポーネントが影響を受けます。

MAP のサイズは、MAP 内で宣言されたコンポーネントの合計サイズです。

UNION 宣言で確立されるデータ域のサイズは、その UNION で定義された最大 MAP のサイズです。

MAP 宣言または **UNION** 構造体にある *parameter_stmt* は、最も近いこれを囲むプログラム単位、インターフェース本体、またはサブプログラムの有効範囲単位内でエンティティーを宣言します。

MAP 宣言にある **%FILL** フィールドは、レコード構成で目的に合ったデータの位置合わせを行うためのプレースホルダーとして使用されます。その他の非充てんコンポーネント、またはデータ域を **%FILL** フィールドと共用するその他の map 宣言内のコンポーネントの部分は未定義です。

UNION 宣言内の少なくとも 1 つの **MAP** にある *component_def_stmts* にデフォルトの初期化が指定されている場合、初期化の最後のオカレンスがコンポーネントの最終初期化になります。

レコード構造内の UNION MAP 宣言のいずれかにデフォルトの初期化が指定されている場合、デフォルトで割り当てられたストレージ・クラスを持つタイプの変数には、以下のいずれかのストレージ・クラスが与えられます。

- **-qsave=defaultinit** または **-qsave=all** オプションのいずれかが指定されている場合、静的ストレージ・クラス
- **-qnosave** オプションが指定されている場合、自動ストレージ・クラス

常に 1 つの MAP のみが共用ストレージに関連付けられます。別の MAP からのコンポーネントが参照される場合、関連した MAP が関連解除され、そのコンポーネントは未定義になります。これで、参照されている MAP がストレージに関連させられます。

map_dcl のコンポーネントが UNION 内のその他の *map_dcl* の **%FILL** コンポーネントに完全にまたは部分的にマップされている場合、そのコンポーネントがデフォルトの初期化か、または割り当てステートメントによって初期化されない限り、オーバーラップ部分の値は未定義です。

UNION および MAP の例

例 1: UNION のサイズは、その UNION 内の最大 MAP のサイズと同じです

```
structure /S/
   union
     map
       integer*4  i, j, k
       real*8  r, s, t
     end map
     map
       integer*4  p, q
       real*4  u, v
     end map
   end union         ! Size of the union is 36 bytes.
 end structure
 record /S/ r
```

例 2: 別の -qsave オプションおよびサブオプションを使用すると、UNION MAP の結果は異なります

```
PROGRAM P
 CALL SUB
 CALL SUB
```

```
END PROGRAM P

SUBROUTINE SUB
  LOGICAL, SAVE :: FIRST_TIME = .TRUE.
  STRUCTURE /S/
    UNION
      MAP
        INTEGER I/17/
      END MAP
      MAP
        INTEGER J
      END MAP
    END UNION
  END STRUCTURE
  RECORD /S/ LOCAL_STRUCT
  INTEGER LOCAL_VAR

  IF (FIRST_TIME) THEN
    LOCAL_STRUCT.J = 13
    LOCAL_VAR = 19
    FIRST_TIME = .FALSE.
  ELSE
    ! Prints " 13" if compiled with -qsave or -qsave=all
    ! Prints " 13" if compiled with -qsave=defaultinit
    ! Prints " 17" if compiled with -qnosave
    PRINT *, LOCAL_STRUCT%j
    ! Prints " 19" if compiled with -qsave or -qsave=all
    ! Value of LOCAL_VAR is undefined otherwise
    PRINT *, LOCAL_VAR
  END IF
END SUBROUTINE SUB
```

例 3: UNION 構造内の MAP 宣言にあるデフォルトの初期化の最後のオカレンスは、コンポーネントの最終初期化になります

```
structure /st/
  union
    map
      integer  i /3/, j /4/
      union
        map
          integer  k /8/, l /9/
        end map
      end union
    end map
    map
      integer  a, b
      union
        map
          integer  c /21/
        end map
      end union
    end map
  end union
end structure
```

```
record /st/ R
print *, R.i, R.j, R.k, R.l      ! Prints "3 4 21 9"
print *, R.a, R.b, R.c           ! Prints "3 4 21"
end
```

例 4: 次のプログラムは **-qintsize=4** および **-qalign=struct=pack** でコンパイルされます。 **UNION MAP** 内のコンポーネントは位置合わせされてパックされます

```
structure /s/
  union
    map
      integer*2  i /z'1a1a'/, %FILL, j /z'2b2b'/
    end map
    map
      integer  m, n
    end map
  end union
end structure
record /s/ r

print '(2z6.4)', r.i, r.j      ! Prints "1A1A  2B2B"
print '(2z10.8)', r.m, r.n     ! Prints "1A1A0000  2B2B0000" however
                               ! the two bytes in the lower order are
                               ! not guaranteed.
r.m = z'abc00cba'              ! Components are initialized by
                               ! assignment statements.
r.n = z'02344320'

print '(2z10.8)', r.m, r.n     ! Prints "ABC00CBA  02344320"
print '(2z6.4)', r.i, r.j      ! Prints "ABC0  0234"
end
```

IBM 拡張 の終り

タイプなしリテラル定数

IBM 拡張

XL Fortran では、タイプなし定数は組み込みタイプを持ちません。 16 進数、8 進数、2 進数、およびホレリス定数は、組み込みリテラル定数を使用する場合ならどのような状況においても使用できます。ただし、これらの定数はタイプ宣言ステートメントでの長さ指定には使用できません (タイプなしの定数は、**CHARACTER** タイプ宣言ステートメントの *type_param_value* では使用できます)。 16 進、8 進、2 進の各定数で認識される桁数は、その定数が使用されるコンテキストによって異なります。

16 進定数

16 進定数の形式は、次のとおりです。

```
►►──┬─X─┬──┬─'─hexadecimal_number─'─┬──┬──────────────────────►◄
    │ └─Z─┘  └─"─hexadecimal_number─"─┘  │
    ├─┬─'─hexadecimal_number─'─┬──┬─X─┬─┤
    │ └─"─hexadecimal_number─"─┘  └─Z─┘ │
    └─Z─hexadecimal_number──────────────┘
```

hexadecimal_number
: 数字 (0-9) と文字 (A-F、a-f) で構成されています。対応する大文字と小文字は等しく扱われます。

16 進定数の **Z***nn...nn* という形式は、スラッシュで区切ってデータ初期化値としてのみ使用できます。 16 進定数のこの形式が **PARAMETER** 属性で事前に定義した定数の名前と同じストリングである場合、XL Fortran は、そのストリングを名前付き定数として認識します。

2x 個の 16 進数が存在する場合、x バイトで表されます。

XL Fortran による定数の解釈方法については、66 ページの『タイプなし定数の使用方法』を参照してください。

16 進定数の例

```
Z'0123456789ABCDEF'
Z"FEDCBA9876543210"
Z'0123456789aBcDeF'
Z0123456789aBcDeF   ! This form can only be used as an initialization value
```

8 進定数

8 進定数の形式は、次のとおりです。

```
►►──┬─O──┬─'─octal_number─'─┬─┬────────────────────────────►◄
    │    └─"─octal_number─"─┘ │
    └─┬─'─octal_number─'─┬──O─┘
      └─"─octal_number─"─┘
```

octal_number
: 数字 (0-7) で構成されるストリングです。

8 進数の 1 桁は 3 ビットで、データ・オブジェクトは 8 ビットの倍数なので、8 進定数のビット数がデータ・オブジェクトに必要なビット数よりも大きくなる場合があります。たとえば、**INTEGER(2)** データ・オブジェクトは、最左端の桁が 0 または 1 の場合、6 桁の 8 進定数によって表されます。**INTEGER(4)** データ・オブジェクトは、最

左端の桁が、0、1、2、3 のいずれかである場合、11 桁の定数によって表されます。**INTEGER(8)** は、最左端の桁が 0 または 1 の場合、22 桁の定数によって表されます。

XL Fortran による定数の解釈方法については、66 ページの『タイプなし定数の使用方法』を参照してください。

8 進定数の例

```
O'01234567'
"01234567"O
```

2 進定数

2 進定数の形式は、次のとおりです。

```
►►──┬─B───┬─'─binary_number─'─┬──┬────────────────────────────────►◄
    │     └─"─binary_number─"─┘  │
    └─┬─'─binary_number─'─┬──B───┘
      └─"─binary_number─"─┘
```

binary_number 0 と 1 の数字で構成されるストリングです。

8x 個の 2 進数が存在する場合、x バイトで表されます。

XL Fortran による定数の解釈方法については、66 ページの『タイプなし定数の使用方法』を参照してください。

2 進定数の例

```
B"10101010"
'10101010'B
```

ホレリス定数

ホレリス定数の形式は、次のとおりです。

```
►►──n──H──character_string────────────────────────────────────────►◄
```

ホレリス定数は、nH に続く、プロセッサーで表示可能な空ではない文字ストリングで構成されています。ここでは、n は、H の後に続く文字の数を示す符号なしの正の整数です。n には、kind 型付きパラメーターを指定することはできません。ストリング中の文字数は、1 ～ 255 文字になります。

注: nH を指定し、n 個未満の文字を n の後に指定すると、入力行を右マージンまで拡張するために使用されるブランクがホレリス定数の一部と見なされます。ホレリス定数は、継続行で継続することができます。ホレリス定数には、n 個以上の文字を使用してください。

-qnoescape コンパイラー・オプションが指定されていない場合は、XL Fortran は、ホレリス定数内のエスケープ・シーケンスも認識します。ホレリス定数にエスケープ・シーケンスが含まれている場合、n は、ソース・ストリング内の文字数ではなく、そのストリングの内部表現での文字数を示します。 (たとえば、2H\"\" は、2 つの二重引用符を意味するホレリス定数を表します。)

XL Fortran は、文字定数、ホレリス定数、**H** 編集記述子、文字ストリング編集記述子、および注釈の中のマルチバイト文字をサポートします。このサポートは、**-qmbcs** オプションによって提供されます。ストリング全体を保持するには短すぎる変数にマルチバイト文字が入っている定数を割り当てると、マルチバイト文字の内部で切り捨てが起こることがあります。

Unicode の文字およびファイル名もサポートします。環境変数 **LANG** が **UNIVERSAL** に設定され、**-qmbcs** コンパイラー・オプションが指定されている場合、コンパイラーは Unicode の文字およびファイル名の読み取りや書き込みを行うことができます。

XL Fortran による定数の解釈方法については、『タイプなし定数の使用方法』を参照してください。

タイプなし定数の使用方法

タイプなし定数のデータ型と長さは、タイプなし定数を使用するコンテキストによって決まります。 XL Fortran は、使用前には変換を行いません。

- **-qctyplss** コンパイラー・オプションを指定してプログラムをコンパイルすると、文字初期化式は、ホレリス定数に適用される規則に従います。
- タイプなし定数は、組み込みデータ型のうちの 1 つだけをとります。
- タイプなし定数を算術単項演算子または論理単項演算子と併用すると、その定数には、デフォルトの整数タイプが設定されます。
- タイプなし定数を算術 2 進演算子、論理 2 進演算子または 2 進関係演算子と併用すると、定数はもう一方のオペランドのデータ型と同じデータ型になります。両方のオペランドがタイプなし定数の場合、関係演算子の両方のオペランドがホレリス定数でない限り、オペランドのデータ型は、デフォルトの整数タイプをとります。このような場合、両方のオペランドとも文字データ型をとります。
- タイプなし定数を連結で使用すると、定数のデータ型は文字データ型となります。
- タイプなし定数を割り当てステートメントの式の右側で使用すると、定数のタイプは、左側の変数のタイプになります。
- タイプなし定数を、特定のデータ型が要求されるコンテキストで使用すると、定数のデータ型は、その特定のデータ型になります。

- タイプなし定数を **DATA** ステートメント、**STATIC** ステートメント、またはタイプ宣言ステートメントの初期値として、あるいは **PARAMETER** ステートメントの名前付き定数の定数値として使用するとき、またタイプなし定数を文字以外のタイプのデータとして扱うときは、次の規則が適用されます。
  - 16 進定数、8 進定数、または 2 進定数が予定された長さより短い場合、XL Fortran は左側にゼロを追加します。予定された長さより長い場合は、コンパイラーは左側で切り捨てを行います。
  - ホレリス定数が予定された長さより短い場合、コンパイラーは右側にブランクを追加します。予定された長さより長い場合は、コンパイラーは右側で切り捨てを行います。
  - タイプなし定数が、長さを継承した文字データ型で名前付き定数の値を指定している場合、名前付き定数の長さは、タイプなし定数で指定したバイト数と等しくなります。
- タイプなし定数を文字タイプのオブジェクトとして扱う場合 (**DATA**、**STATIC**、タイプ宣言、またはコンポーネント定義ステートメントの初期値として使用する、あるいは名前付き定数の定数値として使用する場合は除く)、その長さは、タイプなし定数で指定されたバイト数によって決まります。
- タイプなし定数を複素定数の一部として使用すると、定数のデータ型は複素定数のもう一方の部分のデータ型になります。両方の部分ともタイプなし定数である場合には、定数のデータ型は、両方のタイプなし定数を表現するのに十分な長さを持つ実数データ型になります。
- タイプなし定数を実引き数として使用する場合、対応する仮引き数のタイプは、組み込みデータ型になります。仮引き数は、プロシージャー、ポインター、配列、派生型のオブジェクト、または選択戻り指定子にはなりません。
- タイプなし定数を実引き数として使用し、かつ次の事項が該当する場合、
  - プロシージャーが総称組み込みプロシージャーを参照している
  - すべての引き数がタイプなし定数である
  - 総称プロシージャー名と同じ名前の特定の組み込みプロシージャーが*存在*する

  総称名への参照は、特定のプロシージャーによって解決されます。
- タイプなし定数を実引き数として使用し、かつ次の事項が該当する場合、
  - プロシージャーが総称組み込みプロシージャーを参照している
  - すべての引き数がタイプなし定数である
  - 総称プロシージャー名と同じ名前の特定の組み込みプロシージャーが*存在しない*

  タイプなし定数は、デフォルトの整数に変換されます。特定の組み込み関数が整数引き数をとる場合、その参照はその特定関数によって解決されます。特定の組み込み関数が存在しない場合は、その参照は総称関数によって解決されます。
- タイプなし定数を実引き数として使用し、かつ次の事項が該当する場合、
  - プロシージャーが総称組み込みプロシージャーを参照している
  - タイプなし定数ではない別の引き数が指定されている

タイプなし定数は、その引き数タイプとなります。ただし、コンパイラー・オプション **-qport=typlssarg** を指定した場合は、実引き数がデフォルトの整数に変換されます。選択された特定の組み込みプロシージャーはこのタイプを基にしています。

- タイプなし定数を実引き数として使用し、かつプロシージャー名を総称名として設定しているが、組み込みプロシージャーではない場合、総称プロシージャーの参照は、1 つの特定のプロシージャーによって解決しなければなりません。定数のデータ型は、その特定のプロシージャーの対応する仮引き数のデータ型になります。たとえば、次のようになります。

```
INTERFACE SUB
  SUBROUTINE SUB1( A )
     REAL A
  END SUBROUTINE
  SUBROUTINE SUB2( A, B )
     REAL A, B
  END SUBROUTINE
  SUBROUTINE SUB3( I )
     INTEGER I
  END SUBROUTINE
END INTERFACE
CALL SUB('C0600000'X, '40066666'X)  ! Resolves to SUB2

CALL SUB('00000000'X)               ! Invalid - ambiguous, may
                                    ! resolve to either SUB1 or SUB3
```

- タイプなし定数を実引き数として使用し、かつプロシージャー名を特定名としてのみ設定する場合、定数のデータ型は、対応する仮引き数のデータ型となります。
- タイプなし定数を実引き数として使用し、かつ次の事項が該当する場合、
  - プロシージャー名が総称名または特定名のどちらとしても設定されていない
  - 参照によって定数が渡された

  その定数がホレリス定数でない場合はデフォルトの整数サイズとなりますが、データ型は想定されません。ホレリス定数を渡すためのデフォルトでは、文字実引き数と同様になります。ただし、コンパイラー・オプション **-qctyplss=arg** を使用すると、ホレリス定数は整数実引き数であるかのように渡されます。プロシージャー名を総称名または特定名として設定する方法に関する詳細は、 206 ページの『プロシージャー参照の解決』を参照してください。
- タイプなし定数を実引き数として使用し、かつ次の事項が該当する場合、
  - プロシージャー名が総称名または特定名のどちらとしても設定されていない
  - 値によって定数が渡された

  定数は 16 進定数、2 進定数、および 8 進定数のデフォルト整数の場合と同様に渡されます。

  定数がホレリス定数で、かつデフォルト整数のサイズ未満の場合、XL Fortran は右側にブランクを追加します。定数がホレリス定数で、かつ 8 バイトを超える場合、XL

Fortran は、右側のホレリス定数を切り捨てます。プロシージャー名を総称名または特定名として設定する方法に関する詳細は、 206 ページの『プロシージャー参照の解決』を参照してください。

- タイプなし定数をその他のコンテキストで使用すると、定数のデータ型は、デフォルトの整数タイプとなります。この場合、ホレリス定数は例外となります。ホレリス定数を次の状況で使用した場合、データ型は文字データ型となります。
  - H 編集記述子
  - ホレリス定数である両オペランドとともに関係演算
  - 入出力リスト
- タイプなし定数をデフォルトの整数として扱おうとして、その値がデフォルト整数の値の範囲内に表現できない場合、その定数は、表現可能な種類にプロモートされます。
- kind 型付きパラメーターは、**-qintlog** がオンの場合でも論理定数と置き換えることができず、**-qctyplss** がオンの場合でも文字定数と置き換えることができません。また、タイプなし定数にすることもできません。

式の中のタイプなし定数の例

```
INT=B'1'          ! Binary constant is default integer
RL4=X'1'          ! Hexadecimal constant is default real
INT=INT + O'1'    ! Octal constant is default integer
RL4=INT + B'1'    ! Binary constant is default integer
INT=RL4 + Z'1'    ! Hexadecimal constant is default real
ARRAY(O'1')=1.0   ! Octal constant is default integer

LOGICAL(8) LOG8
LOG8=B'1'         ! Binary constant is LOGICAL(8), LOG8 is .TRUE.
```

IBM 拡張 の終り

タイプの決め方

個々のユーザー定義関数または名前付きエンティティーには、データ型があります。(ホスト関連付けまたは使用関連付けによりアクセスされるエンティティーのタイプは、それぞれ、ホスト有効範囲単位内で決定されるか、またはアクセスされるモジュール内で決定されます。) 名前のタイプは、次の 3 つの方法のいずれかにより、次の順序に従って決定されます。

1. 次の 2 つのいずれかによって明示的に決める方法
   - 指定したタイプ宣言ステートメントから決める方法 (詳細については、 481 ページの『タイプ宣言』を参照してください。)
   - 関数の結果の場合は、指定したタイプ・ステートメントまたはその **FUNCTION** ステートメントから決める方法

2. **IMPLICIT** タイプ・ステートメントから暗黙的に決める方法 (詳細については、 389 ページの『IMPLICIT』を参照してください。)
3. 事前に定義された規則によって暗黙的に決める方法。デフォルトにより (つまり、**IMPLICIT** タイプ・ステートメントがない場合)、名前の最初の文字が I、J、K、L、M、または N の場合は、タイプはデフォルトの整数になります。それ以外の場合は、タイプはデフォルトの実数となります。

特定の有効範囲単位内では、英字、ドル記号、下線を **IMPLICIT** ステートメントに指定していない場合、使用する暗黙のタイプは、ホスト有効範囲単位で使用されている暗黙のタイプと同じになります。プログラム単位およびインターフェース本体は、事前に定義された規則を記述する **IMPLICIT** ステートメントがあるホストと同様に扱われます。

リテラル定数のデータ型は、その形式によって決められます。

変数の定義状況

変数は常に定義済み、または未定義のどちらかで、その定義状況をプログラムの実行中に変更することができます。名前付き定数は、値をもっています。名前付き定数をプログラムの実行時に定義または再定義することはできません。

文字タイプまたは複素数タイプの配列 (セクションを含む)、構造体、および変数は、0 以上のサブオブジェクトからなるオブジェクトです。変数とサブオブジェクト間、および異なる変数のサブオブジェクト間に関連付けを確立することができます。

- すべてのサブオブジェクトが定義されることによって、オブジェクトが定義されます。つまり、それぞれのオブジェクトまたはサブオブジェクトは値を持っていますが、オブジェクトまたはサブオブジェクトが未定義になるまで、または別の値によって再定義されるまで、その値は変わりません。
- オブジェクトが未定義の場合、1 つ以上のサブオブジェクトが未定義です。未定義のオブジェクトや、サブオブジェクトの値は保証されません。

DATA ステートメント、タイプ宣言ステートメント、**STATIC** ステートメントで初期値を持つように指定された変数は、最初は定義済みです。さらに、デフォルトの初期化によって、変数の初期値定義される場合があります。ゼロ・サイズの配列およびゼロ長の文字オブジェクトは、常に定義済みです。

それ以外の変数はすべて、最初は未定義です。

定義を発生させるイベント

以下に示すイベントは、変数を定義済みにします。

1. マスクされた配列割り当てステートメント以外の組み込み割り当てステートメント F95 または **FORALL** 割り当てステートメント F95 を実行すると、等号の前の変数は定義済みになります。

定義済みの割り当てステートメントを実行すると、等号の前の全部または一部の変数が定義済みになる場合があります。

2. マスクされた配列割り当てステートメント F95 、または **FORALL** 割り当てステートメント F95 を実行すると、割り当てステートメントの配列エレメントの一部またはすべてが定義済みになる場合があります。
3. 1 つの入力ステートメントを実行すると、入力ファイルから値を割り当てられた変数はそれぞれ、データを受け取った時点で定義済みになります。単位指定子で内部ファイルを識別する **WRITE** ステートメントを実行すると、書き込まれる各レコードが定義済みになります。

   非同期入力ステートメントを実行すると、対応する **WAIT** ステートメントが実行されるまで変数は定義済みになりません。
4. **DO** ステートメントを実行すると、 **DO** 変数 (もしあれば) は定義済みになります。

Fortran 95

5. さらに、デフォルトの初期化によって、変数が最初から定義される場合があります。

Fortran 95 の終り

6. I/O ステートメント内の暗黙 **DO** リストで指定した処理の実行を開始すると、暗黙 **DO** 変数は定義済みになります。
7. **ASSIGN** ステートメントを実行すると、ステートメント内の変数は、ステートメント・ラベルの値によって定義済みとなります。
8. 仮引き数が **INTENT(OUT)** を持っておらず、それに対応する実引き数全体がステートメント・ラベル以外の値で定義されている場合、プロシージャーに対する参照によって、仮引き数のデータ・オブジェクト全体が定義されます。

   仮引き数に対応する実引き数の対応サブオブジェクトが定義済みの場合、プロシージャーに対する参照によって、**INTENT(OUT)** を持たない仮引き数のサブオブジェクトは定義済みになります。
9. **IOSTAT=** 指定子が入っている I/O ステートメントを実行すると、指定された整数変数は定義済みになります。
10. **SIZE=** 指定子が入っている **READ** ステートメントを実行すると、指定された整数変数は定義済みになります。

IBM 拡張

11. **ID=** 指定子が入っている、XL Fortran の **READ** または **WRITE** ステートメントを実行すると、指定された整数変数は定義済みになります。

12. **DONE=** 指定子が入っている XL Fortran の **WAIT** ステートメントを実行すると、指定された論理変数は定義済みになります。
13. **NUM=** 指定子が入っている、XL Fortran の同期 **READ** または **WRITE** ステートメントを実行すると、指定された整数変数は定義済みになります。

    NUM= 指定子が入っている非同期 **READ** または **WRITE** ステートメントを実行すると、指定された整数変数は定義済みになります。整数変数は、対応する **WAIT** ステートメントの実行時に定義されます。

IBM 拡張 の終り

14. **INQUIRE** ステートメントを実行すると、エラー条件が存在しない場合に、そのステートメントの実行値に値を割り当てられている変数はすべて定義済みになります。
15. 1 つの文字記憶単位が定義済みになると、それに関連するすべての文字記憶単位が定義済みとなります。

    数値記憶単位が定義済みになると、同じタイプの関連した数値記憶単位はすべて定義済みとなります。ただし、**ASSIGN** ステートメントが実行される場合には、**ASSIGN** ステートメント内の変数に関連した変数は、未定義となります。**DOUBLE PRECISION** タイプのエンティティーが定義済みになると、その全体が関連した倍精度実数タイプのエンティティーはすべて定義済みとなります。

    デフォルト以外の整数タイプ、デフォルトまたは倍精度以外の実数タイプ、デフォルト以外の論理タイプ、デフォルト以外の複素数タイプ、任意の長さを持つデフォルト以外の文字タイプ、または非順序タイプのポインターなしのスカラー・オブジェクトは、各ケースごとに異なる未指定の単一記憶単位を占有します。他のポインターと、タイプ、種類、ランクの少なくとも 1 つが明らかに異なるポインターは、未指定の単一記憶単位を占有します。未指定の記憶単位が定義済みとなると、関連するすべての未指定記憶単位は、定義済みとなります。
16. デフォルトの複素数エンティティーが未定義になると、その一部が関連したデフォルトの実数エンティティーは、すべて未定義となります。
17. 一部が関連するデフォルトの実数タイプ・エンティティーまたは複素数タイプ・エンティティーが定義済みとなることにより、デフォルトの複素数エンティティーの両方の部分が定義済みとなり、そのデフォルトの複素数タイプ・エンティティーが定義済みとなります。
18. 部分的に関連したオブジェクトが定義済みとなることにより、数値順序構造体または文字順序構造体のすべてのコンポーネントが定義済みとなり、その構造体は定義済みとなります。
19. **STAT=** 指定子付きの **ALLOCATE** ステートメントまたは **DEALLOCATE** ステートメントを実行すると、 **STAT=** 指定子で指定された変数は、定義済みとなります。
20. ゼロ・サイズの配列を割り当てると、その配列は定義済みとなります。

21. プロシージャーを呼び出すと、そのプロシージャー内のゼロ・サイズの自動オブジェクトは、すべて定義済みとなります。
22. 定義済みのターゲットに、ポインターを関連させるポインター割り当てステートメントを実行すると、そのポインターは定義済みとなります。
23. 非ポインターの割り振り不可自動オブジェクトを含むプロシージャーを呼び出すと、そのオブジェクトの非ポインター・デフォルト初期化済みサブコンポーネントはすべて定義済みになります。
24. 非ポインターの割り振り不可 **INTENT(OUT)** 仮引き数を含むプロシージャーを呼び出すと、そのオブジェクトの非ポインター・デフォルト初期化済みサブコンポーネントはすべて定義済みになります。
25. 非ポインター・コンポーネントがデフォルトの初期化によって初期化される派生型のオブジェクトを割り振ると、コンポーネントとそのサブオブジェクトは定義済みになります。

Fortran 95

26. Fortran 95 で使用される **FORALL** ステートメントまたは構造体では、*index-name* 値セットが評価されると、 *index-name* は定義済みになります。

Fortran 95 の終り

IBM 拡張

27. **COPYIN** 文節にない **THREADPRIVATE** 非ポインター割り振り不可変数が最初の並列領域に入るときに定義される場合、変数のそれぞれの新しいスレッドのコピーが定義されます。
28. **COPYIN** 文節にない **THREADPRIVATE** 共通ブロックが最初の並列領域に入るときに定義される場合、変数のそれぞれの新しいスレッドのコピーが定義されます。
29. **COPYIN** 文節に指定された **THREADPRIVATE** 変数の場合、それぞれの新しいスレッドが、マスター・スレッドの定義、割り振り、およびこの変数の関連付け状況を複写します。したがって、並列領域に入る時に変数のマスター・スレッドのコピーが定義される場合は、変数のそれぞれの新しいスレッドのコピーも定義されます。
30. **COPYIN** 文節内にある **THREADPRIVATE** 共通ブロックの場合、それぞれの新しいスレッドはマスター・スレッドの定義、割り振り、および共通ブロック内の変数の関連付け状況を複写します。したがって、並列領域に入るときに共通ブロック変数のマスター・スレッドのコピーが定義される場合、共通ブロック変数のそれぞれの新しいスレッドのコピーも定義されます。
31. 変数が **PARALLEL**、**PARALLEL DO**、 **DO**、**PARALLEL SECTIONS**、**PARALLEL WORKSHARE**、 **SECTIONS**、または **SINGLE** ディレクティブの **FIRSTPRIVATE** 文節で指定されると、それぞれの新しいスレッドがマスター・ス

レッドの定義とその変数の関連付け状況を複写します。したがって、並列領域に入る時に変数のマスター・スレッドのコピーが定義される場合は、変数のそれぞれの新しいスレッドのコピーも定義されます。

32. 各変数、または共通ブロック内の変数が **COPYPRIVATE** 文節で指定されている場合、 **SINGLE** 構造体に囲まれたコードの実行後で、チーム内のスレッドが構造体から離れる前に変数のすべてのコピーが以下のように定義されます。

    - 変数が **POINTER** 属性を持っている場合、チーム内のその他のスレッドにある変数のコピーは、**SINGLE** 構造体に囲まれたコードを実行したスレッドに属する変数のコピーと同じポインター関連付け状況になります。
    - 変数が **POINTER** 属性を持っていない場合、チーム内のその他のスレッドにある変数のコピーは、**SINGLE** 構造体に囲まれたコードを実行したスレッドに属する変数のコピーと同じ定義になります。

IBM 拡張 の終り

未定義を発生させるイベント

以下に示すイベントは、変数を未定義にします。

1. 指定したタイプの変数が定義済みとなると、関連した別のタイプの変数はすべて未定義になります。しかし、デフォルトの実数タイプの変数がデフォルトの複素数タイプの変数に部分的に関連する場合、実数タイプの変数が定義済みになっていると、複素数タイプの変数は未定義になりません。また、複素数タイプの変数が定義済みになっていると、実数タイプの変数は未定義にはなりません。デフォルトの複素数タイプの変数が別のデフォルトの複素数タイプの変数と部分的に関連していると、一方を定義しても他方が未定義になることはありません。
2. **ASSIGN** ステートメントを実行すると、そのステートメント内の変数は、整数として未定義になります。その変数に関連した変数も未定義になります。
3. 関数の評価によって、関数の引き数、またはモジュールや共通ブロック内の変数が定義済みとなる場合、しかも関数に対する参照が、関数の値によって式の値を決める必要のない式に現れる場合、その引き数または変数は式の計算時に未定義になります。
4. サブプログラム内の **RETURN** ステートメントまたは **END** ステートメントの実行で、再帰的に呼び出す場合は、その有効範囲単位に対してローカルである変数、またはその有効範囲単位の現在のインスタンスに対してローカルである変数のすべてが未定義になります。ただし、以下のものは除きます。
    a. **SAVE** または **STATIC** 属性を持つ変数
    b. 無名共通ブロック内の変数
    c. Fortran 90 に従い、名前付き共通ブロック内の変数で、サブプログラムにあるもの、また、サブプログラムに対する直接参照または間接参照を行う 1 つ以上の

他の有効範囲単位内にあるもの変数がスレッド・ローカル共通ブロックの一部でない限り、▶ IBM XL Fortran はこれらの変数を未定義にはしません。 IBM ◀

d. ホスト有効範囲単位からアクセスする変数
e. Fortran 90 に従い、モジュールからアクセスされる変数で、サブプログラムに対する直接参照または間接参照を行う 1 つ以上の他の有効範囲単位によって、直接または間接的にも参照されるもの ▶ IBM XL Fortran は、これらの変数を未定義にはしません。 IBM ◀
f. Fortran 90 に従い、名前付き共通ブロック内の変数で、最初に定義されていてそれ以降定義または再定義されていないもの ▶ IBM XL Fortran は、これらの変数を未定義にはしません。 IBM ◀

5. 入力ステートメントの実行時にエラー条件またはファイルの終わり条件が発生すると、入力リスト、あるいはそのステートメントの名前リスト・グループで指定した変数は、すべて未定義になります。
6. I/O ステートメントの実行時にエラー条件、ファイルの終わり条件、またはレコードの終わり条件が発生し、そのステートメントに暗黙 **DO** リストが含まれていた場合は、そのステートメント内の暗黙 **DO** 変数はすべて未定義になります。
7. 定義済みの割り当てステートメントを実行すると、等号の前の変数の一部または全部を未定義のままにします。
8. 事前に作成されていないレコードを指定する直接アクセス・ステートメントを実行すると、そのステートメントの入力リストで指定した変数がすべて未定義になります。
9. **INQUIRE** ステートメントを実行すると、変数 **NAME=**、**RECL=**、**NEXTREC=** は未定義になります。
10. 文字記憶単位が未定義になると、関連したすべての文字記憶単位は未定義になります。

    数値記憶単位が未定義になると、関連したすべての数値記憶単位は未定義になります。ただし、異なるタイプの関連した数値記憶単位を定義したことにより、未定義になった場合を除きます (前述の (1) 参照)。

    倍精度の実数タイプのエンティティーが未定義になると、それに全体が関連している倍精度の実数タイプ・エンティティーはすべて、未定義になります。

    未指定の記憶単位が未定義になると、関連した未指定の記憶単位がすべて、未定義になります。
11. プロシージャーに対する参照は、実引き数の対応する部分がステートメント・ラベルの値で定義されている場合、仮引き数の一部を未定義にします。

12. 割り振り可能エンティティーの割り振りが解除されると、そのエンティティーは未定義になります。デフォルトの初期化が定義されていない、サイズがゼロ以外のオブジェクトに対して **ALLOCATE** ステートメントが正常に実行されると、そのオブジェクトは未定義になります。
13. エラー条件が存在する場合に、**INQUIRE** ステートメントを実行すると、**IOSTAT=** 指定子 (もしあれば) 内の変数を除くすべての照会指定子変数が未定義になります。
14. プロシージャーが呼び出されると、以下のように未定義になります。
    a. 実引き数に関連していないオプションの仮引き数は、未定義になります。
    b. **INTENT(OUT)** を持つ非ポインター仮引き数およびそれに関連した実引き数は未定義になります。ただし、デフォルト初期化を持つ非ポインター直接コンポーネントを除きます。
    c. **INTENT(OUT)** を持つポインター仮引き数およびそれに関連した実引き数の関連付け状況は、未定義です。
    d. 実引き数の対応するサブオブジェクトが未定義の場合には、仮引き数のサブオブジェクトは未定義になります。
    e. 関数結果変数は、未定義になります。ただし、関数結果変数が **STATIC** 属性で宣言され、かつ前回の呼び出しで定義されていた場合を除きます。
15. ポインターの関連付け状況が未定義になるか、解除されると、そのポインターは未定義になります。

Fortran 95

16. Fortran 95 では **FORALL** ステートメントまたは構造体の実行が完了すると、*index-name* は未定義になります。

Fortran 95 の終り

IBM 拡張

17. 変数が **PARALLEL**、**PARALLEL DO**、 **DO**、**PARALLEL SECTIONS**、**PARALLEL WORKSHARE**、 **SECTIONS**、または **SINGLE** ディレクティブの **PRIVATE** または **LASTPRIVATE** 文節のいずれかで指定されると、スレッドの最初の作成時に変数のそれぞれの新しいスレッドのコピーが未定義になります。
18. 変数が **PARALLEL**、**PARALLEL DO**、 **DO**、**PARALLEL SECTIONS**、**PARALLEL WORKSHARE**、 **SECTIONS**、または **SINGLE** ディレクティブの **FIRSTPRIVATE** 文節で指定されると、それぞれの新しいスレッドがマスター・スレッドの定義とその変数の関連付け状況を複写します。したがって、並列領域に入る時に変数のマスター・スレッドのコピーが未定義になっている場合は、変数のそれぞれの新しいスレッドのコピーも未定義になります。

19. **INDEPENDENT** ディレクティブの **NEW** 文節で変数が指定されると、その変数は続く **DO** ループの反復の先頭にくるごとに未定義になります。

20. 非同期入力で変数が指定されると、その変数は未定義になり、対応する **WAIT** ステートメントが見つかるまで未定義のままになります。

21. **THREADPRIVATE** 共通ブロックまたは **THREADPRIVATE** 変数が **COPYIN** 文節で指定された場合、それぞれの新しいスレッドはマスター・スレッドの定義、割り振り、およびその変数の関連付け状況を複写します。したがって、並列領域に入る時に変数のマスター・スレッドのコピーが未定義になっている場合は、変数のそれぞれの新しいスレッドのコピーも未定義になります。

22. **THREADPRIVATE** 共通ブロックまたは **THREADPRIVATE** 変数が **ALLOCATABLE** 属性を持つ場合、作成された各コピーの割り振り状況は、現在、割り振りが行われていない状況になります。

23. **THREADPRIVATE** 共通ブロックまたは **THREADPRIVATE** 変数が **POINTER** 属性を持っており、この初期関連付け状況が、デフォルトの初期化または明示的な初期化による関連解除である場合、それぞれのコピーの関連付け状況は関連解除になります。そうでない場合、それぞれのコピーの関連付け状況は未定義です。

24. **THREADPRIVATE** 共通ブロックまたは **THREADPRIVATE** 変数が **ALLOCATABLE** 属性と、**POINTER** 属性のどちらも持っておらず、最初にデフォルトまたは明示的な初期化によって定義された場合、それぞれのコピーは同じ定義を持ちます。そうでない場合、それぞれのコピーは未定義です。

IBM 拡張 の終り

割り振り状況

割り振り可能オブジェクトの割り振り状況は、プログラム実行時に次のいずれかになります。

- 現在、割り振られていない。これはオブジェクトに対して割り振りが行われたことがないこと、またはオブジェクトに対して行われた最新の操作が割り振りの解除であったことを意味します。
- 現在、割り振り済み。これは、オブジェクトが **ALLOCATE** ステートメントによって割り振られ、その後に割り振りが解除されていないことを意味します。
- 未定義。これはオブジェクトが **SAVE** または **STATIC** 属性を持たず、どの有効範囲単位からもアクセスされることなく **RETURN** または **END** ステートメントが実行され、その時点で割り振り済みであったことを意味します。

IBM 拡張

XL Fortran では、この状況は、 **-qxlf90=noautodealloc** オプションの使用時にだけ可

能です。 (たとえば、**xlf90** コンパイル・コマンドの使用時など。)

IBM 拡張 の終り

割り振り可能オブジェクトの割り振り状況が、現在、割り振り済み状況である場合、そのオブジェクトを参照したり、定義することができます。現在、割り振られていない割り振り可能オブジェクトを参照または定義することはできません。割り振り可能オブジェクトの割り振り状況が未定義の場合、そのオブジェクトに対して、参照、定義、割り振り、または割り振り解除を行うことはできません。

割り振り可能オブジェクトの割り振り状況が変更されると、それに応じて、関連した割り振り可能オブジェクトの割り振り状況も変更されます。

IBM 拡張

XL Fortran では、そのようなオブジェクトの割り振り状況は、現在、割り振りが行われている状況のままになります。

IBM 拡張 の終り

Fortran 95

Fortran 95 では、モジュールの有効範囲で宣言されている割り振り可能オブジェクトが **SAVE** 属性を持たず、モジュールを参照する有効範囲単位が実行されずに **RETURN** または **END** ステートメントが実行され、その時点でオブジェクトが割り振り済みであった場合、このオブジェクトの割り振り状況はプロセッサー依存になります。

Fortran 95 の終り

変数のストレージ・クラス

IBM 拡張

注: この項では、変数のストレージ・クラスについてのみ扱っています。名前付き定数およびそのサブオブジェクトは、*リテラル* のストレージ・クラスを持っています。

基本ストレージ・クラス

すべての変数は、基本的には、次の 5 つのストレージ・クラスのいずれか 1 つによって表されます。

Automatic
プロシージャー内の変数のためのもので、プロシージャーの終了時に保持されません。変数はスタック・ストレージに常駐します。

Static
プログラムが終了するまでメモリーを保持する変数のためのもので

す。変数は、データ・ストレージに常駐します。サイズの大きい、初期化されていない変数は、.bss ストレージに常駐します。

Common
共通ブロック変数のためのものです。共通ブロック変数が初期化されると、ブロック全体がデータ・ストレージに常駐します。初期化されない場合は、ブロック全体が .bss ストレージに常駐します。

Controlled Automatic
自動オブジェクトのためのものです。変数はスタック・ストレージに常駐します。 XL Fortran は、プロシージャーに対するエントリーにストレージを割り振り、プロシージャーが完了すると、そのストレージの割り振りを解除します。

Controlled
割り振り可能オブジェクトのためのものです。変数は、動的ストレージに常駐します。ストレージの割り振りおよび割り振り解除は、明示的に行わなければなりません。

2 次ストレージ・クラス

以下のストレージ・クラスは、それ自体にストレージはありませんが、実行時に基本ストレージに関連付けられます。

Pointee
対応する整数ポインターの値によって異なります。

Reference parameter
仮引き数で、対応する実引き数がデフォルトの引き渡し方式または **%REF** によってプロシージャーに渡されます。

Value parameter
仮引き数で、対応する実引き数が値によってプロシージャーに渡されます。

引き渡し方法の詳細については、 197 ページの『%VAL および %REF』を参照してください。

ストレージ・クラスの割り当て

変数名には、次の 3 つの方法によってストレージ・クラスが割り当てられます。

1. 明示
   - 仮引き数は、reference parameter または value parameter の明示的なストレージ・クラスを持ちます。詳細については、 197 ページの『%VAL および %REF』を参照してください。
   - Pointee 変数は、pointee の明示的なストレージ・クラスを持ちます。
   - **STATIC** 属性が明示的に指定された変数は、static の明示的なストレージ・クラスを持ちます。
   - **AUTOMATIC** 属性が明示的に指定された変数は、automatic の明示的なストレージ・クラスを持ちます。

- **COMMON** ブロックに指定された変数は、common の明示的なストレージ・クラスを持ちます。
- **SAVE** 属性が明示的に指定された変数は、static の明示的なストレージ・クラスを持ちます。ただし、これは、変数が **COMMON** ステートメントに対しても指定されない場合に限られます。この場合、ストレージ・クラスは common となります。
- **DATA** ステートメントに指定される変数、またはタイプ宣言ステートメントで初期化される変数は、static のストレージ・クラスを持ちます。ただし、これは、変数が **COMMON** ステートメントに対しても指定されない場合に限られます。この場合、そのストレージ・クラスは common となります。
- 文字タイプまたは派生型である関数結果変数は、reference parameter の明示的なストレージ・クラスを持ちます。
- **SAVE** 属性または **STATIC** 属性のどちらも持っていない関数結果変数は、automatic の明示的なストレージ・クラスを持ちます。
- 自動オブジェクトは、controlled automatic の明示的なストレージ・クラスを持ちます。
- 割り振り可能オブジェクトは controlled という明示的なストレージ・クラスを持ちます。

上記のいずれにも該当しない変数で、明示的なストレージ・クラスを持つ変数と同等でない変数は、明示的なストレージ・クラスを継承します。

上記のいずれにも該当しない変数で、明示的なストレージ・クラスを持つ変数と同等なものは、以下の場合、明示的なストレージ・クラスを持ちます。

- リストを持たない **SAVE** ステートメントが有効範囲単位に存在する
- 変数がメインプログラムの指定部分で宣言されている

2. 暗黙

変数が明示的なストレージ・クラスを持たない場合、次のような暗黙のストレージ・クラスが割り当てられます。

- **IMPLICIT STATIC** ステートメントに指定される英字、ドル記号、下線のいずれかによってその名前が始まる変数は、static のストレージ・クラスを持ちます。
- **IMPLICIT AUTOMATIC** ステートメントに指定される英字、ドル記号、下線のいずれかによってその名前が始まる変数は、automatic のストレージ・クラスを持ちます。

特定の有効範囲単位内で、英字、ドル記号、下線のいずれかが **IMPLICIT STATIC** ステートメントまたは **IMPLICIT AUTOMATIC** ステートメントに指定されていない場合、暗黙のストレージ・クラスはホストと同じになります。

モジュールの指定部分で宣言された変数は、静的ストレージ・クラスに関連させられます。

上記のいずれにも該当しない変数で、暗黙のストレージ・クラスを持つ変数と同等なものは、暗黙のストレージ・クラスを継承します。

3. デフォルト

それ以外の変数はすべて、デフォルトのストレージ・クラスを持ちます。

- **qsave=all** コンパイラー・オプションを指定する場合は、static となります。
- 派生型の変数にデフォルトの初期化が指定されている場合は static になり、そうではなく **-qsave=defaultinit** コンパイラー・オプションを指定する場合は automatic となります。
- **-qnosave** コンパイラー・オプションを指定する場合は、automatic となります。これはデフォルト設定です。

  呼び出しコマンドに関するデフォルト設定値の詳細については、「ユーザーズ・ガイド」の『**-qsave** オプション』を参照してください。

IBM 拡張 の終り

第 4 章 配列の概念

Fortran 90 および Fortran 95 は、プログラマーが配列を操作するための機能セット (一般に配列言語という) を提供します。この章では、配列および配列言語について説明します。

- 『配列』
- 85 ページの『配列宣言子』
- 86 ページの『明示的形状配列』
- 89 ページの『想定形状配列』
- 90 ページの『据え置き形状配列』
- 92 ページの『想定サイズ配列』
- 94 ページの『配列エレメント』
- 95 ページの『配列セクション』
- 102 ページの『配列コンストラクター』
- 105 ページの『配列にかかわる式』

関連情報:

- 285 ページの『第 10 章 ステートメントおよび属性』にある多くのステートメントには、配列についての特別な機能と規則があります。
- この章では、**DIMENSION** ステートメントを頻繁に使用します。 334 ページの『DIMENSION』を参照してください。
- 組み込み関数の多くが、特に配列用です。これらの関数は、主に 541 ページの『変換組み込み関数』として分類されているものです。

配列

配列とは、スカラー・データの順序付けられた列のことです。配列のエレメントはすべて、同一のタイプと型付きパラメーターを持ちます。

全体配列 は配列の名前によって示されます。

```
! In this declaration, the array is given a type and dimension
REAL, DIMENSION(3) :: A
! In these expressions, each element is evaluated in each expression
PRINT *, A, A+5, COS(A)
```

全体配列は、名前付きの定数または変数のいずれかです。

次元の境界

配列内の各次元には、上限および下限があります。これらの境界によって、その次元の添え字として使用できる値の範囲が決められます。次元の境界は、正、負、またはゼロのいずれかの値をとります。

IBM 拡張

XL Fortran では、次元の境界は、-(2**31) から 2**31-1 の範囲内で、正、負、またはゼロのいずれかの値をとります。64 ビット・モードの次元の境界範囲は、 -(2**63) から 2**63-1 です。

IBM 拡張 の終り

下限が対応する上限よりも大きい場合、その配列は*ゼロ・サイズ配列* です。これは、エレメントを持ちませんが、配列の特性は持っています。このような次元の下限は 1、上限は 0 となります。

配列宣言子で境界が指定される場合、

- 下限は宣言式です。省略される場合、デフォルト値として 1 をとります。
- 上限は宣言式またはアスタリスク (*) で、デフォルト値はありません。

関連情報:

111 ページの『宣言式』
609 ページの『LBOUND (ARRAY, DIM)』
688 ページの『UBOUND (ARRAY, DIM)』

次元のエクステント

次元のエクステント とは、その次元内のエレメントの数のことで、上限の値から下限の値を引いて 1 を加えることによって求められます。

```
INTEGER, DIMENSION :: X(5)     ! Extent = 5
REAL :: Y(2:4,3:6)             ! Extent in 1st dimension = 3
                               ! Extent in 2nd dimension = 4
```

下限が上限より大きい次元では、最小のエクステントはゼロとなります。

IBM 拡張

配列内のエレメントの理論上の最大数は、 32 ビット・モードでは 2**31-1 エレメント、または XL Fortran 64 ビット・モードでは 2**63-1 エレメントとなります。ハードウェアのアドレッシングを考慮すると、全体の大きさ (バイト数) がこの値を超えるデータ・オブジェクトの組み合わせを宣言することは現実的ではありません。

IBM 拡張 の終り

共通、等価、引き数のいずれかの関連付けに関連した異なる配列宣言子は、異なるランクとエクステントを持ちます。

配列のランク、形状、およびサイズ

配列の*ランク* とは、配列が持つ次元の数のことです。

```
INTEGER, DIMENSION (10) :: A     ! Rank = 1
REAL, DIMENSION (-5:5,100) :: B  ! Rank = 2
```

Fortran 95 では、1 から 7 までの次元を持つことができます。

IBM 拡張

XL Fortran では、配列は 1 から 20 までの次元を持つことができます。

IBM 拡張 の終り

スカラーは、ランク 0 と見なされます。

配列の*形状* は、ランクおよびエクステントから派生します。 これは、エレメントが対応する次元のエクステントを表す、ランク 1 の配列で表すことができます。

```
INTEGER, DIMENSION (10,10) :: A          ! Shape = (/ 10, 10 /)
REAL, DIMENSION (-5:4,1:10,10:19) :: B   ! Shape = (/ 10, 10, 10 /)
```

配列の*サイズ* とは、配列内のエレメントの数のことで、全次元のエクステントの積に等しくなります。

```
INTEGER A(5)            ! Size = 5
REAL B(-1:0,1:3,4)      ! Size = 2 * 3 * 4 = 24
```

関連情報:
- これらの例では、すべての境界が定数である簡単な配列のみを示しています。より複雑な種類の配列に関してこれらの特性の値を計算する指示については、以下の項を参照してください。
- 関連している組み込み関数としては、 666 ページの『SHAPE(SOURCE)』および 671 ページの『SIZE (ARRAY, DIM)』があります。配列 A のランクは SIZE(SHAPE(A)) です。

配列宣言子

配列宣言子は、配列の形状を宣言するものです。

すべての名前付きの配列について宣言が必要であり、有効範囲単位内で 1 つの名前に対して複数の配列宣言子を入れることはできません。配列宣言子は、 **COMMON**、整数 **POINTER**、**STATIC**、**AUTOMATIC**、**DIMENSION**、**ALLOCATABLE**、**POINTER**、**TARGET** ステートメント、および型 (タイプ) 宣言のいずれかで指定できます。

たとえば、次のようになります。

```
DIMENSION :: A(1:5)         ! Declarator is "(1:5)"
REAL, DIMENSION(1,1:5) :: B ! Declarator is "(1,1:5)"
INTEGER C(10)               ! Declarator is "(10)"
```

ポインターは、スカラー、想定形状配列、または明示的形状配列となることができます。

配列宣言子の形式は、次のとおりです。

```
►►──(──array_spec──)──────────────────────────────────────────────►◄
```

array_spec
: 配列の仕様です。これは配列の下限と上限を設定する次元宣言子のリストで、かつ実行時に一方または両方の境界が設定されることを示します。 各次元には、1 つの次元宣言子が必要です。

IBM 拡張

XL Fortran では、配列は 1 から 20 までの次元を持つことができます。

IBM 拡張 の終り

array_spec は次のいずれかです。

explicit_shape_spec_list
assumed_shape_spec_list
deferred_shape_spec_list
assumed_size_spec

各 *array_spec* は、次の項で説明する異なる種類の配列を宣言します。

明示的形状配列

明示的形状配列とは、境界が各次元で明示的に指定されている配列のことです。

Explicit_shape_spec_list

```
   ┌─,──────────────────────────────┐
   ▼                                │
►►──┬─────────────────┬──upper_bound─┴──────────────────────────────►◄
    └─lower_bound──:──┘
```

lower_bound、*upper_bound*
宣言式です。

いずれかの境界が定数ではない場合、その配列はサブプログラム内で宣言してください。定数以外の境界はサブプログラムへの入り口で決められます。 下限が省略されると、そのデフォルト値は 1 となります。

ランクは、指定された上限の数です。明示的形状仮引き数の形状は、対応する実引き数の形状と異なる場合があります。

サイズは、指定された境界によって決められます。

明示的形状配列の例

```
INTEGER A,B,C(1:10,-5:5)  ! All bounds are constant
A=8; B=3
CALL SUB1(A,B,C)
END
SUBROUTINE SUB1(X,Y,Z)
  INTEGER X,Y,Z(X,Y)      ! Some bounds are not constant
END SUBROUTINE
```

自動割り付け配列

自動割り付け配列は、サブプログラムで宣言される明示的形状配列で、仮引き数やポインティング先配列ではありません。自動割り付け配列は、定数以外の宣言式である 1 つ以上の境界を持ちます。境界は、サブプログラムの入り口で計算され、サブプログラムの実行時は変更されません。

```
INTEGER X
COMMON X
X = 10
CALL SUB1(5)
END

SUBROUTINE SUB1(Y)
  INTEGER X
  COMMON X
  INTEGER Y
  REAL Z (X:20, 1:Y)         ! Automatic array.  Here the bounds are made
                             ! available through dummy arguments and common
                             ! blocks, although Z itself is not a dummy
END SUBROUTINE               ! argument.
```

関連情報:
自動データ・オブジェクトの一般的な情報は、 29 ページの『自動オブジェクト』および 78 ページの『変数のストレージ・クラス』を参照してください。

整合配列

整合 配列とは、サブプログラムで宣言され、定数以外の宣言式である 1 つ以上の境界を持つ明示的形状配列のことをいいます。割り振り可能配列は、仮引き数でなければなりません。

```
SUBROUTINE SUB1(X, Y)
INTEGER X, Y(X*3)  ! Adjustable array.  Here the bounds depend on a
                   ! dummy argument, and the array name is also passed in.
END SUBROUTINE
```

ポインティング先配列

IBM 拡張

ポインティング先配列 は、整数 **POINTER** ステートメントまたはその他の仕様ステートメント内で宣言される明示的形状配列または想定サイズ配列です。

配列がサブプログラム内で宣言されている場合、ポインティング先配列の宣言子には、変数のみを入れることができます。また、その変数は、仮引き数、共通ブロックのメンバー、使用またはホストに関連していなければなりません。 次元のサイズはサブプログラムの入り口で計算され、サブプログラムの実行中は一定の状態に保たれます。

「ユーザーズ・ガイド」に説明されているように、**-qddim** コンパイラー・オプションを使用すると、変数を配列宣言子内にのみ入れることができるという制限が解除され、メインプログラム内の宣言子に変数名を入れることができるようになります。また、配列が参照されるたびに、指定された定数以外の境界がすべて再計算されるため、境界式で使用される変数の値を単に変更するだけでポインティング先配列の特性を変更できます。

```
@PROCESS DDIM
INTEGER PTE, N, ARRAY(10)
POINTER (P, PTE(N))
N = 5
P = LOC(ARRAY(2))  !
PRINT *, PTE       ! Print elements 2 through 6 of ARRAY
N = 7              ! Increase the size
PRINT *, PTE       ! Print elements 2 through 8 of ARRAY
END
```

関連情報:

434 ページの『POINTER (整数)』

IBM 拡張 の終り

想定形状配列

想定形状配列とは、各次元のエクステントが関連した実引き数からとられる仮引き数配列のことです。想定形状配列の名前は、仮引き数であるため、サブプログラム内で宣言してください。

lower_bound 宣言式です。

それぞれの下限は、1 にデフォルト設定されるか、明示的に指定することができます。各上限は、指定された下限に対して (実引き数配列の下限ではない) その次元のエクステントを加え、1 を引いて、サブプログラムの入り口で設定されます。

どの次元のエクステントも、関連した実引き数の対応する次元のエクステントです。

ランクは、*assumed_shape_spec_list* 内のコロンの数です。

形状は、関連した実引き数配列から想定されます。

サイズは、宣言されたサブプログラムの入り口で決められ、関連した引き数配列のサイズと等しくなります。

注: 仮引き数として想定形状配列を持つサブプログラムには、明示インターフェースが必要です。

想定形状配列の例

```
INTERFACE
  SUBROUTINE SUB1(B)
    INTEGER B(1:,:,10:)
  END SUBROUTINE
END INTERFACE
INTEGER A(10,11:20,30)
CALL SUB1 (A)
END
SUBROUTINE SUB1(B)
INTEGER B(1:,:,10:)
! Inside the subroutine, B is associated with A.
! It has the same extents as A but different bounds (1:10,1:10,10:39).
END SUBROUTINE
```

据え置き形状配列

据え置き形状配列とは、割り振り可能配列または配列ポインターのことで、境界をプログラムの実行中に定義または再定義できます。

Deferred_shape_spec_list

各次元のエクステント (ならびに関連する境界、形状、サイズの特性) は、配列が割り振られるか、ポインターが定義済みの配列に関連するまでは未定義です。 適切な照会機能に対する引き数としての場合を除いて、それ以前に配列のどの部分に関しても定義することや参照することはできません。 その時点で、配列ポインターは、ターゲット配列の特性と割り振り可能配列の特性が **ALLOCATE** ステートメントに指定されたと見なします。

ランクは、*deferred_shape_spec_list* 内のコロンの数です。

deferred_shape_spec_list は、 *assumed_shape_spec_list* と同様にみえるときもありますが、据え置き形状配列と想定形状配列は、同じものではありません。 据え置き形状配列は **POINTER** 属性または **ALLOCATABLE** 属性のいずれか 1 つを持っていなければなりませんが、想定形状配列は、 **POINTER** 属性を持たない仮引き数でなければなりません。 据え置き形状配列の境界およびそれに関連した実際のストレージは、配列を再割り振りしたり、ポインターを別の配列に関連させることによって変更できますが、想定形状配列の場合、これらの特性は対象となるサブプログラムの実行中には変更されません。

関連情報:

77 ページの『割り振り状況』
142 ページの『ポインターの割り当て』
289 ページの『ALLOCATABLE』
553 ページの『ALLOCATED(ARRAY) または ALLOCATED(SCALAR)』
557 ページの『ASSOCIATED (POINTER, TARGET)』

割り振り可能配列

ALLOCATABLE 属性を持つ据え置き形状配列は、*割り振り可能配列* と呼ばれます。 **ALLOCATE** ステートメントによって、配列に対してストレージが割り振られたときに、その配列の境界と形状が決められます。

```
INTEGER, ALLOCATABLE, DIMENSION(:,:,:) :: A
ALLOCATE(A(10,-4:5,20)) ! Bounds of A are now defined (1:10,-4:5,1:20)
DEALLOCATE(A)
ALLOCATE(A(5,5,5))    ! Change the bounds of A
```

マイグレーションのためのヒント:

使用されるストレージの最小化:

FORTRAN 77 ソース

```
INTEGER A(1000),B(1000),C(1000)
C 1000 is the maximum size
WRITE (6,*) "Enter the size of the arrays:"
READ (5,*) N

   ⋮
DO I=1,N
  A(I)=B(I)+C(I)
END DO
END
```

Fortran 90 または Fortran 95 ソース

```
INTEGER, ALLOCATABLE, DIMENSION(:) :: A,B,C
WRITE (6,*) "Enter the size of the arrays:"
READ (5,*) N
ALLOCATE (A(N),B(N),C(N))

   ⋮
A=B+C
END
```

関連情報:

77 ページの『割り振り状況』

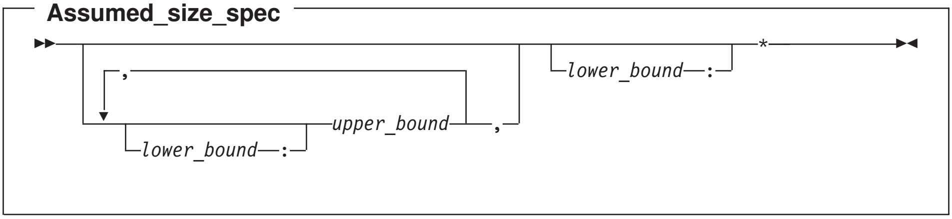
配列ポインター

POINTER 属性を持つ配列は、配列ポインターと呼ばれます。 ポインターの指定または **ALLOCATE** ステートメントの実行によって配列がターゲットと関連したときに、その配列の境界と形状が決められます。 このような配列は、タイプ宣言ステートメント、**POINTER**、または **DIMENSION** ステートメントに指定することができます。

```
REAL, POINTER, DIMENSION(:,:) :: B
REAL, TARGET, DIMENSION(5,10) :: C, D(10:10)
B => C             ! Bounds of B are now defined (1:5,1:10)
B => D             ! B now has different bounds and is associated
                   !   with different storage
ALLOCATE(B(5,5))   ! Change bounds and storage association again
END
```

想定サイズ配列

想定サイズ配列は、仮引き数配列で、関連した実配列からサイズを継承しますが、ランクとエクステントは異なる場合があります。想定サイズ配列は、サブプログラムの中でのみ宣言できます。



lower_bound、*upper_bound*
宣言式です。

いずれかの境界が定数ではない場合、その配列はサブプログラム内で宣言してください。また、定数以外の境界はサブプログラムの入り口で決められます。 下限が省略されると、そのデフォルト値は 1 となります。

最終の次元は上限を持たず、代わりに、アスタリスクによって指定されます。 エレメントに対する参照が実配列の最後を過ぎないようにしてください。

ランクは、その宣言において、*upper_bound* 指定の数に 1 を加えたものに等しくなります。このランクは、それが関連した実配列のランクとは異なることもあります。

サイズは、想定サイズ配列が関連した実引き数から想定されます。

- 実引き数が文字以外の配列である場合、想定サイズ配列のサイズは、実配列のサイズとなります。
- 実引き数が文字以外の配列からの配列エレメントで、このエレメントで始まる配列に残っているサイズが **S** の場合、仮引き数配列のサイズは **S** になります。配列エレメントは、配列エレメントの順序に従って処理されます。
- 実引き数が文字配列、配列エレメント、または配列エレメント・サブストリングのいずれかで、以下を想定した場合、
  - **A** が、文字配列に対する文字ストリング内の開始オフセットである
  - **T** が、元の配列の文字ストリング内の全長である
  - **S** が、仮引き数配列内のエレメントの文字ストリング内の長さである

  仮引き数配列のサイズは、以下のようになります。

  MAX(INT (T - A + 1) / S, 0)

  たとえば、次のようになります。

```
CHARACTER(10) A(10)
CHARACTER(1) B(30)
CALL SUB1(A)           ! Size of dummy argument array is 10
CALL SUB1(A(4))        ! Size of dummy argument array is 7
CALL SUB1(A(6)(5:10)) ! Size of dummy argument array is 4 because there
                       ! are just under 4 elements remaining in A
CALL SUB1(B(12))       ! Size of dummy argument array is 1, because the
                       ! remainder of B can hold just one CHARACTER(10)
END                    ! element.
SUBROUTINE SUB1(ARRAY)
  CHARACTER(10) ARRAY(*)
  ...
END SUBROUTINE
```

想定サイズ配列の例

```
INTEGER X(3,2)
DO I = 1,3
   DO J = 1,2
      X(I,J) = I * J        ! The elements of X are 1, 2, 3, 2, 4, 6
   END DO
END DO
PRINT *,SHAPE(X)            ! The shape is (/ 3, 2 /)
PRINT *,X(1,:)              ! The first row is (/ 1, 2 /)
CALL SUB1(X)
CALL SUB2(X)
END
SUBROUTINE SUB1(Y)
  INTEGER Y(2,*)            ! The dimensions of y are the reverse of x above
  PRINT *, SIZE(Y,1)        ! We can examine the size of the first dimension
                            ! but not the last one.
  PRINT *, Y(:,1)           ! We can print out vectors from the first
  PRINT *, Y(:,2)           ! dimension, but not the last one.
END SUBROUTINE
SUBROUTINE SUB2(Y)
  INTEGER Y(*)              ! Y has a different rank than X above.
  PRINT *, Y(6)             ! We have to know (or compute) the position of
                            ! the last element.  Nothing prevents us from
                            ! subscripting beyond the end.
END SUBROUTINE
```

注:

1. 想定サイズ配列は、それが形状を必要としないサブプログラム参照内での実引き数でない限りは、実行可能な構造体内で全体配列として使用することはできません。

   ```
   ! A is an assumed-size array.
   ```

   ```
   PRINT *, UBOUND(A,1) ! OK - only examines upper bound of first dimension.
   PRINT *, LBOUND(A)   ! OK - only examines lower bound of each dimension.
   ! However, 'B=UBOUND(A)' or 'A=5' would reference the upper bound of
   ! the last dimension and are not allowed.  SIZE(A) and SHAPE(A) are
   ! also not allowed.
   ```

2. 想定サイズ配列のセクションが、その最後のセクションの添え字として、添え字トリプレットを持っている場合、上限を指定する必要があります。 (配列セクションおよび添え字トリプレットについては、続く項で説明します。)

```
! A is a 2-dimensional assumed-size array
PRINT *, A(:, 6)       ! Triplet with no upper bound is not last dimension.
PRINT *, A(1, 1:10)    ! Triplet in last dimension has upper bound of 10.
PRINT *, A(5, 5:9:2)   ! Triplet in last dimension has upper bound of 9.
```

配列エレメント

配列エレメントとは、配列を作成するスカラー・データです。各エレメントは、タイプおよび型付きパラメーター、さらには **INTENT**、 **PARAMETER**、および **TARGET** 属性を親配列から継承します。 **POINTER** 属性は継承しません。

配列エレメント指定子 によって配列エレメントを識別します。配列エレメント指定子の形式は、次のとおりです。

```
>>-+-array_name--------+--(--subscript_list--)-----------------><
   '-array_struct_comp-'
```

array_name
: 配列の名前です。

array_struct_comp
: 構造体のコンポーネントで、その右端の *comp_name* は配列です。

subscript
: スカラー整数式です。

IBM 拡張

subscript は、XL Fortran 内の実数式です。

IBM 拡張 の終り

注

- 添え字の数は、配列内の次元の数と等しくなければなりません。
- *array_struct_comp* がある場合、構造体コンポーネントの各部分は、右端を除いて、ランクがゼロでなければなりません (つまり、配列名や配列セクションであってはなりません)。
- 各添え字式の値は、対応する次元の下限を下回ったり、上限を超えるようなことがあってはなりません。

添え字値 は、各添え字式の値と配列の次元によって異なります。この値によって、配列エレメント指定子で識別すべき配列のエレメントが決まります。

関連情報:

48 ページの『構造体コンポーネント』
100 ページの『配列セクションおよび構造体コンポーネント』

配列エレメントの順序

配列のエレメントは、*配列エレメント順序* という順序で、ストレージ内に並べられています。この順序においては、添え字は最初の次元で最初に変更され、続いて、残りの次元で変更されます。

たとえば、`A(2, 3, 2)` として宣言された配列は、次のエレメントを持ちます。

```
   配列エレメントの位置                 配列エレメントの順序
 -------------------------            --------------------
         A(1,1,1)                              1
         A(2,1,1)                              2
         A(1,2,1)                              3
         A(2,2,1)                              4
         A(1,3,1)                              5
         A(2,3,1)                              6
         A(1,1,2)                              7
         A(2,1,2)                              8
         A(1,2,2)                              9
         A(2,2,2)                             10
         A(1,3,2)                             11
         A(2,3,2)                             12
```

配列セクション

配列セクションは、配列の選択された部分です。配列セクションとは、配列からエレメントを指定したり、その配列の各エレメントから指定の添え字や派生型のコンポーネントを指定する配列サブオブジェクトのことです。配列セクションは、配列でもあります。

注: この導入部では、構造体コンポーネントを含まない簡単なケースについて説明しています。 100 ページの『配列セクションおよび構造体コンポーネント』では、構造体コンポーネントでもある配列セクションの指定方法に関する追加規則について説明しています。

```
▶▶─array_name─(─section_subscript_list─)─┬─────────────────┬──────────▶◀
                                         └─substring_range─┘
```

セクション添え字:

```
▶▶─┬─subscript──────────┬──────────────────────────────────────────▶◀
   ├─subscript_triplet──┤
   └─vector_subscript───┘
```

section_subscript 特定の次元によって、エレメントの一部を指定します。次の組み合わせから構成されています。

subscript

スカラー整数式です。 94 ページの『配列エレメント』 に説明があります。

IBM 拡張

subscript は、XL Fortran 内の実数式です。

IBM 拡張 の終り

subscript_triplet, vector subscript

指定された次元内の添え字の順序 (空の場合もある) を指定します。 詳細については、 97 ページの『添え字トリプレット』および 99 ページの『ベクトル添え字』を参照してください。

注: 配列セクションと配列エレメントを区別するために、次元のどれか 1 つ以上は、添え字トリプレットまたはベクトル添え字でなければなりません。

```
INTEGER, DIMENSION(5,5,5) :: A
A(1,2,3) = 100
A(1,3,3) = 101
PRINT *, A(1,2,3)    ! A single array element, 100.
PRINT *, A(1,2:2,3)  ! A one-element array section, (/ 100 /)
PRINT *, A(1,2:3,3)  ! A two-element array section,
                     ! (/ 100, 101 /)
```

substring_range

```
▶▶─(─┬───────────┬─:─┬───────────┬─)──────────────────────────────▶◀
     └─int_expr1─┘   └─int_expr2─┘
```

int_expr1 および *int_expr2* は、サブストリング式と呼ばれるスカラー整数式で、 40 ページの『文字サブストリング』で定義されています。 この式により、配列セクション内の各エレメントのサブストリングのそれぞれ左端および右端の文字を指定します。 *substring_range* というオプションの項目がある場合、そのセクションは文字オブジェクトの配列からでなければなりません。

配列セクションは、個々の添え字、添え字トリプレット、およびベクトル添え字の値の順序で指定された配列エレメントを桁の順番に並べて形成されます。

たとえば、SECTION = A(1:3, (/ 5,6,5 /), 4) の場合、
最初の次元について、数字は、1、2、3 の順です。
2 番目の次元について、数字は、5、6、5 の順です。
3 番目の次元について、添え字は、定数の 4 です。

セクションは、この順序で、次の A のエレメントから構成されています。

```
A(1,5,4)   |                         |        SECTION(1,1)
A(2,5,4)   |----- First column -----|        SECTION(2,1)
A(3,5,4)   |                         |        SECTION(3,1)
A(1,6,4)     |                         |      SECTION(1,2)
A(2,6,4)     |----- Second column ----|      SECTION(2,2)
A(3,6,4)     |                         |      SECTION(3,2)
A(1,5,4)       |                         |    SECTION(1,3)
A(2,5,4)       |----- Third column -----|    SECTION(2,3)
A(3,5,4)       |                         |    SECTION(3,3)
```

配列セクションの一部を例としてあげます。

```
INTEGER, DIMENSION(10,20) :: A
! These references to array sections require loops or multiple
! statements in FORTRAN 77.
PRINT *, A(1:5,1)                 ! Contiguous sequence of elements
PRINT *, A(1:20:2,10)             ! Noncontiguous sequence of
elements
PRINT *, A(:,5)                   ! An entire column
PRINT *, A( (/1,10,5/), (/7,3,1/) ) ! A 3x3 assortment of elements
```

関連情報:

48 ページの『構造体コンポーネント』。

添え字トリプレット

添え字トリプレットは、2 つの添え字とストライドから構成されています。この添え字トリプレットは、単一の次元での配列エレメント位置に対応する数字の順序を定義します。

```
>>-+-------------+--:--+-------------+--+------------+---------><
   '-subscript1--'     '-subscript2--'  '-:--stride--'
```

subscript1、*subscript2*

次元について、指標の順序列の中での最初と最後の値を指定する添え字です。

subscript1 を省略すると、その次元の下限が使用されます。 subscript2 を省略すると、その次元の上限が使用されます。 (想定サイズ配列のセクションを指定するとき、 subscript2 は最後の次元について必須となります。)

stride

スカラー整数式です。この式で、選択された次のエレメントの添え字を示すために増分する桁数を指定します。

IBM stride は、 XL Fortran 内の実数式です。 IBM

stride を省略すると、1 の値をとります。 stride は、ゼロ以外の値になります。

- 正の stride は、subscript1 から始まり、 subscript2 を超えない最大の整数に至るまで、その stride で指定した値ずつ増分する整数の順序列を指定します。 subscript1 が subscript2 より大きい場合、その順序列は空となります。
- stride が負の場合は、その順序列は、subscript1 で始まり、subscript2 と同じ整数かまたはそれより大きい整数で最小のものまで、 stride によって指定した値ずつ増分します。 subscript2 が subscript1 より大きい場合は、その順序列は空になります。

順序列での値の計算は、 154 ページの『DO ステートメントの実行』で示す手順で行います。

配列セクションの配列エレメントを選択するうえで使用したすべての値が宣言された境界の範囲内にあれば、添え字トリプレット内の添え字は、その次元について宣言された境界の範囲内である必要はありません。

```
INTEGER A(9)
PRINT *, A(1:9:2)  ! Count from 1 to 9 by 2s: 1, 3, 5, 7, 9.
PRINT *, A(1:10:2) ! Count from 1 to 10 by 2s: 1, 3, 5, 7, 9.
                   ! No element past A(9) is specified.
```

添え字トリプレットの例

```
REAL, DIMENSION(10) :: A
INTEGER, DIMENSION(10,10) :: B
CHARACTER(10) STRING(1:100)
```

```
PRINT *, A(:)                ! Print all elements of array.
PRINT *, A(:5)               ! Print elements 1 through 5.
PRINT *, A(3:)               ! Print elements 3 through 10.

PRINT *, STRING(50:100)      ! Print all characters in
                             ! elements 50 through 100.

! The following statement is equivalent to A(2:10:2) = A(1:9:2)
A(2::2) = A(:9:2)            ! LHS = A(2), A(4), A(6), A(8), A(10)
                             ! RHS = A(1), A(3), A(5), A(7), A(9)
                             ! The statement assigns the odd-numbered
                             ! elements to the even-numbered elements.

! The following statement is equivalent to PRINT *, B(1:4:3,1:7:6)
PRINT *, B(:4:3,:7:6)        ! Print B(1,1), B(4,1), B(1,7), B(4,7)

PRINT *, A(10:1:-1)          ! Print elements in reverse order.

PRINT *, A(10:1:1)           ! These two are
PRINT *, A(1:10:-1)          ! both zero-sized.
END
```

ベクトル添え字

ベクトル添え字は、ランク 1 の整数の配列式です。その式のエレメントの値に対応する添え字の順序列を指定します。

IBM ベクトル添え字は、XL Fortran 内の実数配列式です。 IBM

順序列は順番どおりでなくてもかまいません。また、値が重複して入っていることもあります。

```
INTEGER A(10), B(3), C(3)
PRINT *, A( (/ 10,9,8 /) ) ! Last 3 elements in reverse order
B = A( (/ 1,2,2 /) )       ! B(1) = A(1), B(2) = A(2), B(3) = A(2) also
END
```

2 つ以上のエレメントが同じ値を持つベクトル添え字がある配列セクションは、多対 1 セクションと呼ばれます。このようなセクションは、
- 割り当てステートメントの等号の左側に入れてはなりません。
- **DATA** ステートメントによって初期化してはなりません。
- **READ** ステートメント内で入力項目として使用してはなりません。

注:

1. 内部ファイルとして使用される配列セクションには、ベクトル添え字は使えません。
2. ベクトル添え字を持つ配列セクションを実引き数として渡す場合には、関連した仮引き数を定義または再定義することはできません。
3. ベクトル添え字を持つ配列セクションは、ポインター割り当てステートメントのターゲットにはなりません。

```
! We can use the whole array VECTOR as a vector subscript for A and B
INTEGER, DIMENSION(3) :: VECTOR= (/ 1,3,2 /), A, B
INTEGER, DIMENSION(4) :: C = (/ 1,2,4,8 /)
A(VECTOR) = B            ! A(1) = B(1), A(3) = B(2), A(2) = B(3)
A = B( (/ 3,2,1 /) )     ! A(1) = B(3), A(2) = B(2), A(3) = B(1)
PRINT *, C(VECTOR(1:2))  ! Prints C(1), C(3)
END
```

配列セクションおよびサブストリングの範囲

サブストリングの範囲を持つ配列セクションの場合、結果における各エレメントは、その配列セクションの対応するエレメントの指定された文字ストリングです。 右端の配列名またはコンポーネント名は、タイプ文字でなければなりません。

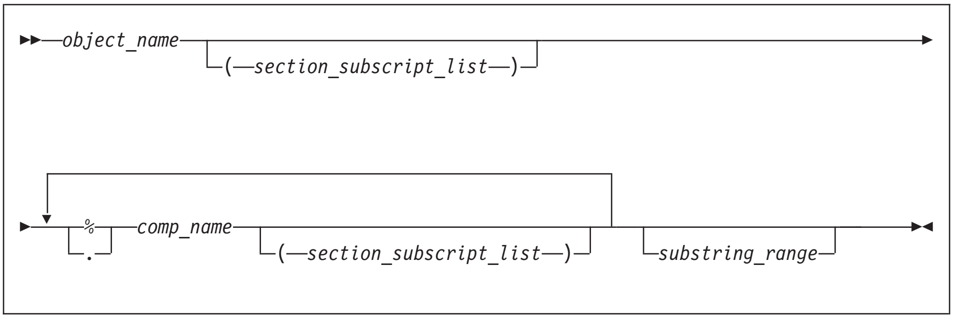
```
PROGRAM SUBSTRING
TYPE DERIVED
  CHARACTER(10) STRING(5)       ! Each structure has 5 strings of 10 chars.
END TYPE DERIVED
TYPE (DERIVED) VAR, ARRAY(3,3)  ! A variable and an array of derived type.

VAR%STRING(:)(1:3) = 'abc'      ! Assign to chars 1-3 of elements 1-5.
VAR%STRING(3:)(4:6) = '123'     ! Assign to chars 4-6 of elements 3-5.

ARRAY(1:3,2)%STRING(3)(5:10) = 'hello'
                                ! Assign to chars 5-10 of the third element in
                                ! ARRAY(1,2)%STRING, ARRAY(2,2)%STRING, and
END                             ! ARRAY(3,2)%STRING
```

配列セクションおよび構造体コンポーネント

配列セクションおよび構造体コンポーネントがどのようにオーバーラップしているかを理解するには、 48 ページの『構造体コンポーネント』の構文について理解しておく必要があります。

配列セクションとしてこの項の始めに定義したものは、考え得る配列セクションのサブセットにすぎません。配列名または *section_subscript_list* を持つ配列名は、構造体コンポーネントのサブオブジェクトであることがあります。

struct_sect_subobj:



object_name 派生型のオブジェクトの名前です。

section_subscript_list、*substring_range*
95 ページの『配列セクション』で定義されているものと同じです。

comp_name 派生型コンポーネントの名前です。

% または . 区切り文字。

注: . (ピリオド) 区切り文字は IBM 拡張です。

注:

1. 最後のコンポーネントのタイプによって配列のタイプが決まります。
2. 構造体コンポーネントの一部分のみがゼロ以外のランクを持つことができます。右端の *comp_name* は、ゼロ以外のランクを持つ *section_subscript_list* を持つか、あるいは別の部分がゼロ以外のランクを持たなければなりません。
3. ゼロ以外のランクを持つ部分から右側のすべての部分では、 **POINTER** 属性を持ってはなりません。

```
TYPE BUILDING_T
  LOGICAL RESIDENTIAL
END TYPE BUILDING_T

TYPE STREET_T
  TYPE (BUILDING_T) ADDRESS(500)
END TYPE STREET_T

TYPE CITY_T
  TYPE (STREET_T) STREET(100,100)
END TYPE CITY_T

TYPE (CITY_T) PARIS
TYPE (STREET_T) S
TYPE (BUILDING_T) RESTAURANT
! LHS is not an array section, no subscript triplets or vector subscripts.
PARIS%STREET(10,20) = S
! None of the parts are array sections, but the entire construct
!   is a section because STREET has a nonzero rank and is not
```

```
!   the rightmost part.
PARIS%STREET%ADDRESS(100) = BUILDING_T(.TRUE.)

! STREET(50:100,10) is an array section, making the LHS an array section
!   with rank=2, shape=(/51,10/).
! ADDRESS(123) must not be an array section because only one can appear
!   in a reference to a structure component.
PARIS%STREET(50:100,10)%ADDRESS(123)%RESIDENTIAL = .TRUE.
END
```

配列セクションのランクおよび形状

構造体コンポーネントのサブオブジェクトではない配列セクションの場合、そのランクは *section_subscript_list* 内の添え字トリプレットおよびベクトル添え字の数です。 形状配列内のエレメントの数は、添え字トリプレットとベクトル添え字の数と同じで、その形状配列内の各エレメントは、対応する添え字トリプレットまたはベクトル添え字によって指定された順序列内の整数値の数です。

構造体コンポーネントのサブオブジェクトである配列セクションの場合、そのランクと形状は、配列名または配列セクションであるコンポーネントの一部のランクおよび形状と同じです。

```
DIMENSION :: ARR1(10,20,100)
TYPE STRUCT2_T
  LOGICAL SCALAR_COMPONENT
END TYPE
TYPE STRUCT_T
  TYPE (STRUCT2_T), DIMENSION(10,20,100) :: SECTION
END TYPE

TYPE (STRUCT_T) STRUCT

! One triplet + one vector subscript, rank = 2.
! Triplet designates an extent of 10, vector subscript designates
!   an extent of 3, thus shape = (/ 10,3 /).
ARR1(:, (/ 1,3,4 /), 10) = 0

! One triplet, rank = 1.
! Triplet designates 5 values, thus shape = (/ 5 /).
STRUCT%SECTION(1,10,1:5)%SCALAR_COMPONENT = .TRUE.

! Here SECTION is the part of the component that is an array,
!   so rank = 3 and shape = (/ 10,20,100 /), the same as SECTION.
STRUCT%SECTION%SCALAR_COMPONENT = .TRUE.
```

配列コンストラクター

配列コンストラクターとは、指定されたスカラー値の順序列のことです。配列コンストラクターは、そのエレメント値が順序列内で指定されたランク 1 の配列を構成します。

```
▶▶─(/─ac_value_list─/)──────────────────────────────────────────▶◀
```

ac_value
: 配列エレメントに値を指定する式または暗黙 **DO** リストです。 配列コンストラクター内の各 *ac_value* は、同じタイプと型付きパラメーターを持たなくてはなりません。

  ac_value は次のようになります。

  - スカラー式の場合、その値は配列コンストラクターのエレメントを指定します。
  - 配列式の場合、式のエレメントの値は、配列エレメントの順に、配列コンストラクターのエレメントの対応する順序列を指定します。
  - 暗黙 **DO** リストの場合、**DO** 構造体内でのように、 *ac_do_variable* の制御のもとに *ac_value* 順序列を形成するために展開されます。

配列コンストラクターのデータ型は、*ac_value_list* 式のデータ型と同じです。配列コンストラクター内のすべての式が定数式である場合、その配列コンストラクターは定数式になります。

1 より大きいランクの配列を構成するには、組み込み関数を使用します。 詳細については、 658 ページの『RESHAPE (SOURCE, SHAPE, PAD, ORDER)』を参照してください。

```
INTEGER, DIMENSION(5) :: A, B, C, D(2,2)
A = (/ 1,2,3,4,5 /)               ! Assign values to all elements in A
A(3:5) = (/ 0,1,0 /)              ! Assign values to some elements
C = MERGE (A, B, (/ T,F,T,T,F /)) ! Construct temporary logical mask

! The array constructor produces a rank-one array, which
!   is turned into a 2x2 array that can be assigned to D.
D = RESHAPE( SOURCE = (/ 1,2,1,2 /), SHAPE = (/ 2,2 /) )

! Here, the constructor linearizes the elements of D in
!   array-element order into a one-dimensional result.
PRINT *, A( (/ D /) )
```

配列コンストラクターの暗黙 DO リスト

配列コンストラクター内の暗黙 **DO** ループは、各エレメントを個々に指定することを避けるために、値の定期的または周期的な順序列を作成するのに役立ちます。

ループによって生成された値の順序列が空の場合、ランクが 1 のゼロ・サイズ配列が形成されます。

```
▶▶─(─ac_value_list─,─implied_do_variable─ = ─expr1─,─expr2─┬──────────┬─)──────────▶◀
                                                            └─,─expr3─┘
```

implied_do_variable

名前付きのスカラー整数または実数の変数です。 IBM XL Fortran では、*implied_do_variable* は実数式です。 IBM 実行不能ステートメントでは、タイプは整数でなければなりません。制限式 *expr1* または *expr2* では、*implied_do_variable* の値を参照しないでください。このループ処理では、 327 ページの『DATA』にある暗黙 **DO** に対する規則と同じ規則に従い、暗黙 **DO** 変数のタイプに基づいて、整数または実際の演算を使用します。

変数は、暗黙 **DO** の有効範囲を持ちますが、指定する配列コンストラクターの暗黙 **DO** の中に別の暗黙 **DO** と同じ名前が入っていてはなりません。

```
M = 0
PRINT *, (/ (M, M=1, 10) /) ! Array constructor implied-DO
PRINT *, M                  ! M still 0 afterwards
PRINT *, (M, M=1, 10)       ! Non-array-constructor implied-DO
PRINT *, M                  ! This one goes to 11
PRINT *, (/ ((M, M=1, 5), N=1, 3) /)
! The result is a 15-element, one-dimensional array.
! The inner loop cannot use N as its variable.
```

expr1、*expr2*、および *expr3*

整数スカラー式です。

IBM 拡張

XL Fortran では、expr1、expr2、および expr3 は実数式です。

IBM 拡張 の終り

```
PRINT *, (/ (I, I = 1, 3) /)
! Sequence is (1, 2, 3)
PRINT *, (/ (I, I = 1, 10, 2) /)
! Sequence is (1, 3, 5, 7, 9)
PRINT *, (/ (I, I+1, I+2, I = 1, 3) /)
! Sequence is (1, 2, 3, 2, 3, 4, 3, 4, 5)

PRINT *, (/ ( (I, I = 1, 3), J = 1, 3 ) /)
! Sequence is (1, 2, 3, 1, 2, 3, 1, 2, 3)

PRINT *, (/ ( (I, I = 1, J), J = 1, 3 ) /)
! Sequence is (1, 1, 2, 1, 2, 3)

PRINT *, (/2,3,(I, I+1, I = 5, 8)/)
```

```
! Sequence is (2, 3, 5, 6, 6, 7, 7, 8, 8, 9).
! The values in the implied-DO loop before
!   I=5 are calculated for each iteration of the loop.
```

配列にかかわる式

配列は、スカラーと同じ種類の式および演算で使用できます。 組み込み演算、割り当て、またはエレメント型プロシージャーは 1 つまたは複数の配列に適用できます。

複数の配列オペランドがかかわる式では、各配列の対応するエレメントを割り当てまたは評価できるように、配列は同じ形状でなければなりません。定義済みオペレーションの場合、配列は異なる形状でもかまいません。 同じ形状を持つ配列間には、*整合性* があります。 整合性のあるエンティティーが予期されるコンテキストでは、スカラー値を使用することもできます。これは、どの配列とも整合性がとれるため、各配列エレメントがスカラーと同じ値を持つことができます。

たとえば、次のようになります。

```
INTEGER, DIMENSION(5,5) :: A,B,C
REAL, DIMENSION(10) :: X,Y
! Here are some operations on arrays
A = B + C         ! Add corresponding elements of both arrays.
A = -B            ! Assign the negative of each element of B.
A = MAX(A,B,C)    ! A(i,j) = MAX( A(i,j), B(i,j), C(i,j) )
X = SIN(Y)        ! Calculate the sine of each element.
! These operations show how scalars are conformable with arrays
A = A + 5         ! Add 5 to each element.
A = 10            ! Assign 10 to each element.
A = MAX(B, C, 5)  ! A(i,j) = MAX( B(i,j), C(i,j), 5 )

END
```

関連情報:

539 ページの『エレメント型組み込みプロシージャー』
127 ページの『組み込み割り当て』
497 ページの『WHERE』では、配列内の一部のエレメントに値を割り当てる方法を示します。
139 ページの『FORALL 構造体』

第 5 章 式および割り当て

この章では、式および割り当てステートメントの構成、解釈、および計算について説明します。

- 『はじめに』
- 109 ページの『定数式』
- 110 ページの『初期化式』
- 111 ページの『宣言式』
- 113 ページの『演算子および式』
- 122 ページの『拡張組み込みおよび定義済み演算』
- 123 ページの『式の計算』
- 127 ページの『組み込み割り当て』
- 131 ページの『WHERE 構造体』
- F95 139 ページの『FORALL 構造体』 F95
- 142 ページの『ポインターの割り当て』

関連情報:

- 179 ページの『定義済み演算子』
- 180 ページの『定義済み割り当て』

はじめに

式とは、データの参照または計算のことで、オペランド、演算子、括弧から構成されます。式を計算すると、タイプ、形状、および場合によっては型付きパラメーターを持った値が生成されます。

オペランド は、スカラーまたは配列のいずれかです。*演算子* は、組み込みまたは定義済みのいずれかです。単項演算の形式は次のとおりです。

演算子 オペランド

2 進演算の形式は次のとおりです。

オペランド$_1$ *演算子 オペランド*$_2$

ここで、2 つのオペランドの形状は整合性がとられます。一方のオペランドが配列で、もう一方がスカラーである場合、そのスカラーは、その配列と同じ形式を持つ配列として扱われ、その配列のすべてのエレメントがスカラーの値を持つことになります。

括弧内の式はどれもデータ・エンティティーとして扱われます。括弧を使用して式の明示的解釈を指定することもできます。さらに、括弧を使用して式の代替形式を制限することもできます。これにより、式の計算時に中間値の大きさと精度を制御できます。例として、次に 2 つの式を示します。

```
(I*J)/K
I*(J/K)
```

この 2 つの式は、数学的には等しいものですが、計算の結果として異なる計算値が出ることもあります。

1 次子

1 次子 は、最も単純な形式の式です。以下のいずれかになります。

- データ・オブジェクト
- 配列コンストラクター
- 構造体コンストラクター
- IBM 複素数コンストラクター IBM
- 関数参照
- 括弧の付いた式

データ・オブジェクトである 1 次子は、想定サイズ配列にはできません。

1 次子の例

```
12.3             ! Constant
'ABCDEFG'(2:3)   ! Subobject of a constant
VAR              ! Variable name
(/7.0,8.0/)      ! Array constructor
EMP(6,'SMITH')   ! Structure constructor
SIN(X)           ! Function reference
(T-1)            ! Expression in parentheses
```

タイプ、パラメーター、および形状

1 次子のタイプ、型付きパラメーター、および形状は、次のように決定されます。

- データ・オブジェクトまたは関数参照では、それぞれにオブジェクトまたは関数参照のタイプ、型付きパラメーター、および形状を必要とします。総称関数参照のタイプ、パラメーターおよび形状は、その実引き数のタイプ、パラメーター、およびランクによって決定されます。
- 構造体コンストラクターはスカラーであり、そのタイプはコンストラクター名のタイプです。
- 配列コンストラクターは、コンストラクター式の数によって決められた形状を持ち、そのタイプおよびパラメーターは、コンストラクター式のタイプおよびパラメーターによって決定されます。
- 括弧で囲んだ式は、その式のタイプ、パラメーター、および形状を要求します。

ポインターがポインター以外の仮引き数に関連した演算に 1 次子として入れられる場合、ターゲットが参照されます。 1 次子のタイプ、パラメーター、および形状は、ターゲットのタイプ、パラメーターおよび形状です。ポインターがターゲットと関連していない場合、そのポインターは、対応する仮引き数がポインターであるプロシージャー参照内の実引き数、またはポインター割り当てステートメント内のターゲットとしてのみ指定されます。

演算 *[expr1] op expr2* の場合、 *op* が単項式であるか、あるいは *expr1* がスカラーであると、その演算の形状は *expr2* の形状となります。そうでない場合、その形状は *expr1* の形状となります。

式のタイプおよび形状は、演算子および式の 1 次子のタイプと形状によって決まります。式のタイプは、組み込みタイプまたは派生型です。組み込みタイプの式は、kind パラメーターを持ち、これが文字タイプの場合には、length パラメーターも持ちます。

定数式

定数式 とは、各演算が組み込みタイプで、各 1 次子が次のいずれかである式のことです。

- 定数または定数のサブオブジェクト
- 各エレメントおよび各暗黙 **DO** の境界とスライドに関して、その 1 次子が、定数式または暗黙 **DO** 変数のいずれかの式である配列コンストラクター
- コンポーネントが定数式である構造体コンポーネント
- 各引き数が定数式であるエレメント型組み込み関数の参照
- 各引き数が定数式である変形可能組み込み関数の参照
- F95 変形可能組み込み関数 **NULL** の参照 F95
- 配列照会関数 (**ALLOCATED** を除く)、数値照会関数、**BIT_SIZE** 関数、**LEN** 関数、または **KIND** 関数に対する参照。各引き数は、定数式であるか、またはその照会した特性が想定されておらず、定数式想定以外の式によっても定義されていない、しかも **ALLOCATE** ステートメントまたはポインター割り当てステートメントによっても定義されていない変数であるかのどちらかです。
- 括弧の付いた定数式

式内の添え字またはサブストリングはすべて、定数式でなければなりません。

定数式の例

```
-48.9
name('Pat','Doe')
TRIM('ABC    ')
(MOD(9,4)**3.5)
```

初期化式

初期化式 は定数式です。定数式の規則は、初期化式にも適用されます。ただし、次の規則によって制約を受ける 1 次子の項目には適用されません。

- 指数演算は、整数の累乗のみが可能です。
- エレメント型組み込み関数の参照できる 1 次子は、各引き数が整数タイプまたは文字タイプの初期化式である、整数タイプまたは文字タイプになります。
- 変形可能関数 **REPEAT**、**RESHAPE**、**SELECTED_INT_KIND**、**SELECTED_REAL_KIND**、**TRANSFER**、または **TRIM** のうちのいずれか 1 つのみを参照することができます。各引き数は、初期化式でなければなりません。次の総称組み込み関数 (および関連する特定関数) を使用することもできます。

IBM 拡張

– ABS (および **ABS**、 **DABS**、および **QABS** 特定関数のみ)
– AIMAG、IMAG
– CONJG
– DIM (および **DIM**、**DDIM**、および **QDIM** 特定関数のみ)
– DPROD
– INT、REAL、DBLE、QEXT、CMPLX、DCMPLX、QCMPLX
– MAX
– MIN
– MOD
– NINT
– SIGN
– INDEX、SCAN、VERIFY (オプションで 3 番目の引き数が使用可能)

IBM 拡張 の終り

– F95 NULL F95

初期化式に、同じ指定箇所に指定されたオブジェクトの配列境界または型付きパラメーターの照会関数に対する参照が入っている場合、その型付きパラメーターまたは配列境界をそれ以前の部分で指定しなければなりません。前の指定部分とは、同じステートメント内の照会関数の左側でもかまいません。

初期化式の例

```
3.4**3
KIND(57438)
(/'desk','lamp'/)
'ab'//'cd'//'ef'
```

宣言式

宣言式は、文字長や配列境界などの項目を指定するために使用する制限を指定した式です。

宣言式 はスカラー、整数、制限式です。

制限式 とは、各演算が組み込みタイプで、各 1 次子が次のような式です。

- 定数または定数のサブオブジェクト
- **OPTIONAL** 属性および **INTENT(OUT)** 属性のいずれも持たない仮引き数である変数、あるいはその変数のサブオブジェクト
- 共通ブロック内にある変数またはその変数のサブオブジェクト
- 使用関連付けまたはホスト関連付けによってアクセス可能な変数またはその変数のサブオブジェクト
- 各エレメントおよび各暗黙 **DO** の境界とストライドに関して、その 1 次子が、制限式または暗黙 **DO** 変数のいずれかの式である配列コンストラクター
- 各コンポーネントが制限式である構造体コンストラクター
- 配列照会関数 (**ALLOCATED** を除く)、ビット照会関数 **BIT_SIZE**、文字照会関数 **LEN**、種類照会関数 **KIND**、または数値照会関数に対する参照。各引き数は、制限式であるか、あるいは、照会した特性が想定サイズ配列の最後の次元の上限に依存しない変数、制限式以外の式で定義されない変数、または **ALLOCATE** ステートメントまたはポインター割り当てステートメントによって定義できない変数のうちのいずれかです。

Fortran 95

- 各引き数が制限式である、本書で定義されているその他の組み込み関数の参照

Fortran 95 の終り

IBM 拡張

- システム照会関数に対する参照。どの引き数も制限式です。

IBM 拡張 の終り

- 添え字式またはサブストリング式はすべて制限式です。
- 宣言関数に対する参照。どの引き数も制限式です。

Fortran 95

宣言関数 は、宣言式内で使用することができます。関数は、組み込み、内部、またはステートメント関数でない純粋な関数である場合、宣言関数です。宣言関数は、仮プロシ

ージャーの引き数を持つことはできず、再帰的にすることもできません。

Fortran 95 の終り

同一の有効範囲単位内の前回の宣言、有効範囲単位に対して現在有効な暗黙の入力規則、あるいはホスト関連付けや使用関連付けによって指定されているものがある場合は、宣言式内の変数は、タイプと型付きパラメーターを持たなければなりません。宣言式内の変数が暗黙型指定規則によって型指定される場合、後続のタイプ宣言ステートメントにその変数が現われるときは常に、暗黙タイプおよび型付きパラメーターを確認しなければなりません。

同一の宣言部分に指定されたエンティティーの配列境界または型付きパラメーターに関する照会関数への参照が宣言式に含まれる場合、型付きパラメーターまたは配列境界を、前の部分で指定しなければなりません。宣言式が、同じ宣言部分で指定された配列のエレメントの値に対する参照を含む場合、その配列境界は、それより前の宣言で指定しなければなりません。前の宣言とは、同じステートメント内の照会関数の左側でもかまいません。

宣言式の例

```
LBOUND(C,2)+6    ! C is an assumed-shape dummy array
ABS(I)*J         ! I and J are scalar integer variables
276/NN(4)        ! NN is accessible through host association
```

Fortran 95

以下の例は、ユーザー定義の関数 fact を、配列値の関数結果の変数内の宣言式で使用する方法について示しています。

```
MODULE MOD
CONTAINS
  INTEGER PURE FUNCTION FACT(N)
  INTEGER, INTENT(IN) :: N
  ...
  END FUNCTION FACT
END MODULE MOD

PROGRAM P
  PRINT *, PERMUTE('ABCD')
  CONTAINS
  FUNCTION PERMUTE(ARG)
     USE MOD
     CHARACTER(*), INTENT(IN) :: ARG
     ...
     CHARACTER(LEN(ARG)) :: PERMUTE(FACT(LEN(ARG)))
     ...
  END FUNCTION PERMUTE
END PROGRAM P
```

Fortran 95 の終り

演算子および式

この項では、最小から最大までの評価優先順位にしたがって式のレベルを表しています。

一般式

一般式の形式 (*general_expr*) は次のとおりです。

```
>>-+----------------------------------+--expr--------------------><
   '-general_expr--defined_binary_op-'
```

defined_binary_op
: 定義済み 2 進演算子です。 122 ページの『拡張組み込みおよび定義済み演算』を参照してください。

expr
: 次に定義する 4 種類の式のうちのいずれかです。

4 種類の組み込み式とは、算術式、文字式、関係式、論理式です。

算術式

算術式 (*arith_expr*) を計算すると、数値が得られます。 *arith_expr* の形式は次のとおりです。

```
>>-+-------------------------+--arith_term-----------------------><
   '-+------------+--+-+-+--'
     '-arith_expr-'  '- - -'
```

arith_term の形式は次のとおりです。

```
>>-+--------------------+--arith_factor--------------------------><
   '-arith_term--+-/-+--'
                 '-*-'
```

arith_factor の形式は次のとおりです。

```
▶▶─arith_primary─┬─────────────────────┬──────────────────────────────────────▶◀
                 └─ ** ─arith_factor─┘
```

arith_primary は算術タイプの 1 次子です。

次の表は、使用可能な算術演算子と、算術式内での各演算子の演算優先順位を示しています。

算術演算子	意味	優先順位
**	指数	1 番目
*	乗算	2 番目
/	除算	2 番目
+	加算または恒等	3 番目
-	減算または否定	3 番目

XL Fortran では、2 つ以上の加算演算子が含まれている算術式の各項は、左から右に向かって計算されます。たとえば、2+3+4 は、プロセッサーが、数学的に同等で括弧が正しい別の方法でこの式を解釈できたとしても、(2+3)+4 のように計算されます。

2 つ以上の乗算演算子または除算演算子が含まれている各項を計算する場合、因数は左から右に向かって計算されます。 たとえば、2*3*4 は (2*3)*4 のように計算されます。

2 つ以上の指数演算子を含んでいる因数が計算される場合、1 次子は右から左に向かって計算されます。たとえば、2**3**4 は 2**(3**4) のように計算されます。 (繰り返しになりますが、数学的に同等であるようになります。)

XL Fortran 演算優先順位の異なる演算子が 2 つ以上含まれている算術式を計算する場合、演算子の優先順位で、計算の順序が決められます。たとえば、式 -A**3 の場合、指数演算子 (**) の方が否定演算子 (-) よりも優先順位が高くなります。したがって、指数演算子のオペランドは、否定演算子のオペランドとして使用される式を形成するために結合されます。すなわち、-A**3 は -(A**3) と同じように計算されます。

A**-B または A*-B のように、式の中で算術演算子を 2 つ続けることはできないことに注意してください。ただし、A**(-B) や A*(-B) という式を使用することはできます。

式で、整数による整数の除算を指定している場合、その結果は、ゼロに近い整数に丸められます。たとえば、(-7)/3 は、-2 の値を持ちます。

IBM 拡張

浮動小数点式の計算時に発生し得る例外条件の詳細については、「ユーザーズ・ガイド」の『*浮動小数点演算例外の検出とトラップ*』を参照してください。

IBM 拡張 の終り

算術式の例

算術式	括弧付きの同等式
-b**2/2.0	-((b**2)/2.0)
i**j**2	i**(j**2)
a/b**2 - c	(a/(b**2)) - c

算術式のデータ型

恒等演算子および否定演算子は 1 つのオペランドに対してのみ演算を行うため、演算結果の値のタイプは、オペランドのタイプと同じになります。

次の表は、算術演算子が 1 対のオペランドに対して演算を行った場合に得られるタイプを示したものです。

表記法: *T(param)*。ここで *T* はデータ型 (I: 整数、 R: 実数、X: 複素数)、*param* は kind 型付きパラメーターです。

表 3. 2 進算術演算子の結果タイプ

第 1 オペランド	第 2 オペランド I(1)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(1)	I(1)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(2)	I(2)	I(2)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(4)	I(4)	I(4)	I(4)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
I(8)	I(8)	I(8)	I(8)	I(8)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
R(4)	R(4)	R(4)	R(4)	R(4)	R(4)	R(8)	R(16)	X(4)	X(8)	X(16)
R(8)	R(8)	R(8)	R(8)	R(8)	R(8)	R(8)	R(16)	X(8)	X(8)	X(16)
R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	R(16)	X(16)	X(16)	X(16)
X(4)	X(4)	X(4)	X(4)	X(4)	X(4)	X(8)	X(16)	X(4)	X(8)	X(16)
X(8)	X(8)	X(8)	X(8)	X(8)	X(8)	X(8)	X(16)	X(8)	X(8)	X(16)
X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)	X(16)

IBM 拡張

注:

1. **-qfloat=rndsngl** を指定しないと、XL Fortran では、 **REAL(4)** の演算は、 **REAL(8)** の内部精度を使用して実行されます。 **-qfloat=rndsngl** を指定すると、 XL Fortran では **REAL(4)** の演算は **REAL(4)** の内部精度を使用して実行されます。この演算の実行を修正する方法の詳細については、「ユーザーズ・ガイド」の『*浮動小数点演算例外の検出とトラップ*』を参照してください。 **REAL(16)** 値は、最も近い値に丸めて使用しなければなりません。丸めモードは、サブプログラムの最初と最後でのみ変更が可能です。これをサブプログラムの呼び出し中に変更することはできません。サブプログラム内で変更した場合は、呼び出しルーチンに制御が戻る前に、復元しなければなりません。
2. XL Fortran は、 **INTEGER(4)** の算術を使用して整数の演算を行います。また、データ項目の長さが 8 バイトの場合は、 **INTEGER(8)** の算術を使用して整数の演算を行います。 **INTEGER(1)** または **INTEGER(2)** データ型を必要とするコンテキストの中で中間結果を使用すると、その結果は必要に応じて変換されます。

```
INTEGER(2) I2_1, I2_2, I2_RESULT
INTEGER(4) I4
I2_1 = 32767              ! Maximum I(2)
I2_2 = 32767              ! Maximum I(2)
I4 = I2_1 + I2_2
PRINT *, "I4=", I4        ! Prints "I4=65534"

I2_RESULT = I2_1 + I2_2  ! Assignment to I(2) variable
I4 = I2_RESULT            ! and then assigned to an I(4)
PRINT *, "I4=", I4        ! Prints "I4=-2"
END
```

IBM 拡張 の終り

文字

文字式を計算すると、文字タイプの結果が得られます。 *char_expr* の形式は次のとおりです。

```
>>-+----------------+--char_primary-------------------------------><
   '-char_expr--//-'
```

char_primary は、文字タイプの 1 次子です。文字式の中の 1 次子はすべて、同じ kind 型付きパラメーターを持ちます。これは、結果の kind 型付きパラメーターでもあります。

文字演算子としては、// しかなく、これは連結を表します。

1 つ以上の連結演算子が含まれている文字式の場合、1 次子がつなげられて、1 つのストリングになります。ストリングの長さは、個々の 1 次子の長さの和に等しくなります。たとえば、 'AB'//'CD'//'EF' は、6 文字のストリング 'ABCDEF' と評価されます。

文字式に括弧を付けても、結果の値は変わりません。

継承した長さの文字ストリングが以下のものの宣言に使用された場合、長さが括弧内のアスタリスクによって宣言された (継承した長さを示します) オペランドの連結を文字式に入れることができます。
- **FUNCTION** ステートメント、**SUBROUTINE** ステートメント、または **ENTRY** ステートメントで指定された仮引き数。仮引き数の長さは、呼び出し中に関連した実引き数の長さを想定します。
- 名前付き定数。これは定数の値の長さをとります。
- 外部関数の結果の長さ。呼び出し側の範囲指定単数は、アスタリスクで関数名を宣言することはできません。呼び出し時には、関数結果の長さは、この定義された長さを想定します。

文字式の例

```
CHARACTER(7)  FIRSTNAME,LASTNAME
FIRSTNAME='Martha'
LASTNAME='Edwards'
PRINT *, LASTNAME//', '//FIRSTNAME      ! Output:'Edwards, Martha'
END
```

関係式

関係式 (*rel_expr*) を指定できるのは、論理式の中だけです。関係式は、算術関係式または文字関係式です。

算術関係式

算術関係式は、2 つの算術式の値を比較します。式の形式は次のとおりです。

```
▶▶—arith_expr1—relational_operator—arith_expr2———————————————————▶◀
```

arith_expr1 と **arith_expr2**
算術式です。複素数タイプの式で指定できるのは、*relational_operator* が **.EQ.**、**.NE.**、<>、==、または /= の場合のみです。

relational_operator
次のいずれかです。

関係演算子	意味
.LT. または <	より小

関係演算子	意味
.LE. または <=	以下
.EQ. または ==	等しい
.NE. または *<> または /=	等しくない
.GT. または >	より大
.GE. または >=	以上

注: * XL Fortran 比較演算子。

演算子で指定された関係をオペランドの各値が満足する場合、その演算関係式の論理値は、`.true.` であると解釈されます。指定された関係をオペランドの値が満足していない場合には、式の論理値は、`.false.` になります。

式のタイプまたは kind 型付きパラメーターが異なる場合、その式の値は計算の前にその式 (*arith_expr1* + *arith_expr2*) のタイプおよび kind 型付きパラメーターに変換されます。

算術関係式の例:

```
IF (NODAYS .GT. 365) YEARTYPE = 'leapyear'
```

文字関係式

文字関係式は、2 つの文字式の値を比較します。式の形式は次のとおりです。

```
►►—char_expr1—relational_operator—char_expr2——————————————————►◄
```

char_expr1 と *char_expr2*
それぞれ文字式です。

relational_operator
117 ページの『算術関係式』で説明した関係演算子です。

どのような関係演算子の場合でも、文字関係式の解釈には、照合順序を使用します。照合順序の低い値を持つ文字式の方が、高い値を持つ文字式よりも小さいと見なされます。文字式での計算は、1 度に 1 文字ずつ、左から右に向かって行われます。組み込み関数 (**LGE**、**LLT**、および **LLT**) を使用して、 ASCII コードによる照合順序で指定された文字ストリングを比較することもできます。どのような関係演算子の場合でも、オペランドの長さが等しくなければ短い方のオペランドの右側にブランクが追加されます。*char_expr1* および *char_expr2* の長さが両方ともゼロの場合、これらは等しいと見なされます。

IBM 拡張

char_expr1 および *char_expr2* が XL Fortran のマルチバイト文字 (MBCS) の場合でも、ASCII コードによる照合順序が使用されます。

IBM 拡張 の終り

文字関係式の例:

```
IF (CHARIN .GT. '0' .AND. CHARIN .LE. '9') CHAR_TYPE = 'digit'
```

論理

論理式 (*logical_expr*) を計算すると、論理タイプの結果が得られます。論理式の形式は次のとおりです。

```
>>-+-----------------------------+-logical_disjunct-------------><
   '-logical_expr--+-.EQV.--+----'
                   +-.NEQV.-+
                   |    (1) |
                   '-.XOR.--'
```

注:

1 XL Fortran 論理演算子 (logical operator)

logical_disjunct の形式は、次のとおりです。

```
>>-+---------------------------+-logical_term-------------------><
   '-logical_disjunct--.OR.-'
```

logical_term の形式は、次のとおりです。

```
>>-+-----------------------+-logical_factor---------------------><
   '-logical_term--.AND.-'
```

logical_factor の形式は、次のとおりです。

```
>>-+------------------------------------+-----------------------><
   '-+-------+--+-logical_primary-+----'
     '-.NOT.-'  '-rel_expr--------'
```

logical_primary は論理タイプの 1 次子です。

rel_expr は関係式です。

論理演算子には、次のものがあります。

論理演算子	意味	優先順位
.NOT.	論理否定	1 番目 (最上位)
.AND.	論理積	2 番目
.OR.	包括的論理和	3 番目
.XOR. (注 * を参照。)	排他的論理和	4 番目 (最下位) (注 * を参照。)
.EQV.	論理等価	4 番目 (最下位)
.NEQV.	論理非等価	4 番目 (最下位)

注: * XL Fortran 論理演算子。

IBM 拡張

.XOR. 演算子は、**-qxlf77=intxor** コンパイラー・オプションを指定した場合にのみ、組み込み演算子として扱われます。 (詳細については、「ユーザーズ・ガイド」の『**-qxlf77** オプション』を参照してください。) それ以外の場合は、定義済み演算子として扱われます。 **.XOR.** を組み込み演算子として扱う場合、総称インターフェースにより、**.XOR.** を拡張することもできます。

IBM 拡張 の終り

優先順位の異なる 2 つ以上の演算子が含まれている論理式を計算する場合、演算子の優先順位によって計算の順序が決められます。たとえば、`A.OR.B.AND.C` という式は、`A.OR.(B.AND.C)` のように計算されます。

論理式の値

`x1` および `x2` が論理値であると仮定した場合、次の表を使って論理式の値を決めます。

x1	**.NOT. x1**
真	偽
偽	真

x1	**x2**	**.AND.**	**.OR.**	**.XOR.**	**.EQV.**	**.NEQV.**
偽	偽	偽	偽	偽	真	偽
偽	真	偽	真	真	偽	真
真	偽	偽	真	真	偽	真
真	真	真	真	偽	真	偽

論理式の値を求めるとき、必ずしも式全体が計算されなくてもよい場合があります。次のような論理式を考えてみてください。(LFCT が論理タイプの関数であるとします。)

```
A .LT. B .OR. LFCT(Z)
```

A が B より小さい場合は、関数参照が計算されなくても、この式が真であることがわかります。

XL Fortran は、n が kind 型付きパラメーターである **LOGICAL(n)** または **INTEGER(n)** の結果に対して、論理式を計算します。 n の値は各オペランドの kind パラメーターによって異なります。

単項論理演算子 **.NOT.** の場合、デフォルトでは、n は kind 型付きパラメーターによって異なります。たとえば、オペランドが **LOGICAL(2)** のとき、結果も **LOGICAL(2)** になります。

次の表に単項演算の結果のタイプを示します。

オペランド	単項演算の結果
* BYTE	INTEGER(1) *
LOGICAL(1)	LOGICAL(1)
LOGICAL(2)	LOGICAL(2)
LOGICAL(4)	LOGICAL(4)
LOGICAL(8)	LOGICAL(8)
* タイプなし	デフォルトの整数 *

注: * XL Fortran 内の単一操作の結果のタイプ

2 つのオペランドの長さが同じである場合は、n はその長さになります。

IBM 拡張

異なる kind パラメーターを持つオペランドが指定された 2 進論理演算の場合、式の kind 型付きパラメーターは 2 つのオペランドのうちの長い方と同じになります。たとえば、1 つのオペランドが **LOGICAL(4)** で、もう一方が **LOGICAL(2)** のとき、結果は **LOGICAL(4)** になります。

IBM 拡張 の終り

次の表に 2 進演算の結果のタイプを示します。

表 4. 2 進論理式の結果のタイプ

	第 2 オペランド					
第 1 オペランド	***BYTE**	**LOGICAL(1)**	**LOGICAL(2)**	**LOGICAL(4)**	**LOGICAL(8)**	***タイプなし**
***BYTE**	*INTEGER(1)	*LOGICAL(1)	*LOGICAL(2)	*LOGICAL(4)	*LOGICAL(8)	*INTEGER(1)
LOGICAL(1)	LOGICAL(1)	LOGICAL(1)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	LOGICAL(1)
LOGICAL(2)	LOGICAL(2)	LOGICAL(2)	LOGICAL(2)	LOGICAL(4)	LOGICAL(8)	LOGICAL(2)
LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(4)	LOGICAL(8)	LOGICAL(4)
LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)	LOGICAL(8)
***タイプなし**	*INTEGER(1)	*LOGICAL(1)	*LOGICAL(2)	*LOGICAL(4)	*LOGICAL(8)	*デフォルトの整数

注: * XL Fortran 内の 2 進論理式の結果のタイプ

式の結果がデフォルトの整数として扱われ、その値がデフォルト整数の値の範囲内で表現できない場合、定数は表現可能な形にされます。

1 次子

1 次子の形式は、次のとおりです。

```
>>-+------------------+--primary-------------------------------><
   '-defined_unary_op-'
```

defined_unary_op 定義済みの単項演算子です。 『拡張組み込みおよび定義済み演算』を参照してください。

拡張組み込みおよび定義済み演算

定義済み演算は、定義済み単項演算または定義済み 2 進演算のいずれかです。これは、関数および総称インターフェース・ブロックによって定義されます (172 ページの『インターフェース・ブロック』を参照してください)。 定義済み演算は、組み込み演算ではありませんが、組み込み演算子を定義済み演算に拡張することはできます。たとえば、加算の組み込み 2 進演算子 (+) の意味を拡張することによって、派生型の 2 つのオブジェクトを加算することができます。拡張組み込み演算子がタイプなしのオペランドを持っている場合、その演算は組み込みとして評価されます。

拡張される単項組み込み演算のオペランドは、組み込み演算子で要求されるタイプを持つことはできません。拡張される 2 進組み込み演算子のオペランドのいずれか、またはその両方は、組み込み演算子で必要とされるタイプおよびランクを持つことはできません。

定義済み演算の定義済み演算子は、総称インターフェースの中で定義されなければなりません。

定義済み演算子は、拡張された組み込み演算子であり、次の形式を持ちます。

注:

1 XL Fortran 定義済み演算子

2 XL Fortran 定義済み演算子

定義済み演算子の文字数は、31 文字まででなければなりません。また、定義済み演算子は、組み込み演算子または論理リテラル定数と同じであってはなりません。

インターフェース・ブロック内の演算子の定義および拡張方法の詳細については、 176 ページの『総称インターフェース・ブロック』を参照してください。

式の計算

演算子の優先順位

式には、複数の種類の演算子を含めることができます。その場合、式は、次のような演算子の優先順位に従って、左から右に向かって計算されます。

1. 定義済み単項演算子
2. 算術演算子
3. 文字演算子
4. 関係演算子
5. 論理演算子
6. 定義済み 2 進演算子

たとえば、次のような論理式があるとします。

```
L .OR. A + B .GE. C
```

ここで、L が論理タイプで A、B、 C が実数タイプだとすると、その式は以下の論理式と同じように計算されます。

```
L .OR. ((A + B) .GE. C)
```

拡張組み込み演算子は、その優先順位を維持します。つまり演算子は、定義済み単項演算子または定義済み 2 進演算子にはなりません。

解釈規則の要約

演算子を使った 1 次子は、次の順序で組み合わされます。

1. 括弧を使用する
2. 演算子の優先順位
3. 因数内の指数を右から左に向かって解釈する
4. 項内の乗算および除算を左から右に向かって解釈する
5. 算術式内の加算および減算を左から右に向かって解釈する
6. 文字式内の連結を左から右に向かって解釈する
7. 論理項内の論理積を左から右に向かって解釈する
8. 論理和内の論理和を左から右に向かって解釈する
9. 論理式内の論理等価を左から右に向かって解釈する

式の計算

算術式、文字式、関係式、論理式は、次の規則に従って計算されます。

- 変数または関数は、その使用時に定義しておく必要があります。整数オペランドは、ステートメント・ラベル値ではなく、整数値で定義してください。文字データ・オブジェクト内で参照される文字または、配列や配列セクション内で参照される配列エレメントは、すべて参照実行時に定義します。構造体のすべてのコンポーネントを、構造体が参照されるときに定義する必要があります。ポインターは、定義済みターゲットと関連します。

  配列エレメントの参照、配列セクションの参照、および添え字の参照を実行するには、そのセクション添え字、およびサブストリング式の計算が必要となります。配列エレメント添え字、セクション添え字、サブストリング式の計算や、配列コンストラクターの暗黙 **DO** の境界およびストライドは影響を受けることはありません。また、配列を含んでいる式のタイプにも影響しません。 105 ページの『配列にかかわる式』を参照してください。計算結果の値が実行可能プログラム内で数学的に定義されていない定数整数演算または浮動小数点演算は使用できません。このような式が定数以外のもので実行された場合、式は実行時に検出されます。 (たとえば、ゼロ除算、またはゼロ値の 1 次子に対するゼロ値または負数値の累乗演算)。または、負数の 1 次子に対して実数レベルでの累乗演算を行うこともできません。

- ステートメント内で関数を呼び出した場合、それが関数参照のあるステートメント内の他のエンティティーの計算に影響を与えたり、そのような計算によって影響を受けるようなことがあってはなりません。ある式の値が真の場合に、論理 **IF** ステートメントまたは **WHERE** ステートメントの式の中で関数参照を呼び出すと、実行されるステートメント内のエンティティーに影響を与える場合があります。関数参照によって関数の実引き数が定義または未定義にされる場合、その引き数または関連するエンティティーを同じステートメント内の他の場所に指定してはなりません。次に例を示します。

```
A(I) = FUNC1(I)
Y = FUNC2(X) + X
```

上記のようなステートメントは、FUNC1 の参照によって I が定義される場合、または FUNC2 の参照によって X が定義される場合には使用できません。

関数の実引き数の計算は、その関数参照を含んでいる式のデータ型によって影響を受けることはありません。また、関数参照を含んでいる式のデータ型は、関数の実引き数の計算によって影響を受けることはありません。

- ステートメント関数参照の引き数をその参照の計算によって変えることはできません。

IBM 拡張

コンパイラー・オプションの中には最終結果のデータ型に影響を与えるものもあります。

- **-qintlog** コンパイラー・オプションを使用すると、式およびステートメント内で整数と論理値を混在させることができます。その計算結果のデータ型および kind 型付きパラメーターは、関係するオペランドと演算子によって異なります。一般的には、以下のようになります。
  - 論理単項演算子 (**.NOT.**) および算術単項演算子 (**+**、**-**) の場合

オペランドのデータ型	**単項演算の結果のデータ型**
BYTE	INTEGER(1)
INTEGER(n)	INTEGER(n)
LOGICAL(n)	LOGICAL(n)
タイプなし	デフォルトの整数

この場合の n は、kind 型付きパラメーターを表します。 n は、 **-qintlog** がオンの場合でも、論理定数と置換できません。また、 **-qctyplss** がオンの場合でも、文字定数と置換できません。さらに n をタイプなし定数にすることもできません。**INTEGER** および **LOGICAL** データ型の場合、結果の長さはオペランドの kind 型付きパラメーターの長さと同じになります。

  - 2 進論理演算子 (**.AND.**、 **.OR.**、 **.XOR.**、 **.EQV.**、 **.NEQV.**) および算術 2 進演算子 (******、 *****、 **/**、 **+**、 **-**) に関して、次の表に結果のデータ型の種類をまとめてあります。

	第 2 オペランド			
第 1 オペランド	**BYTE**	**INTEGER(y)**	**LOGICAL(y)**	**タイプなし**
BYTE	INTEGER(1)	INTEGER(y)	LOGICAL(y)	INTEGER(1)
INTEGER(x)	INTEGER(x)	INTEGER(z)	INTEGER(z)	INTEGER(x)
LOGICAL(x)	LOGICAL(x)	INTEGER(z)	LOGICAL(z)	LOGICAL(x)

第 1 オペランド	第 2 オペランド BYTE	INTEGER(y)	LOGICAL(y)	タイプなし
タイプなし	INTEGER(1)	INTEGER(y)	LOGICAL(y)	デフォルトの整数

注: z は、結果の kind 型付きパラメーターであり、**z** は、**x** と **y** のうちの大きい方に等しくなります。たとえば、**LOGICAL(4)** タイプのオペランドおよび **INTEGER(2)** タイプのオペランドを持つ論理式の結果タイプは、**INTEGER(4)** になります。

2 進論理演算子 (**.AND.**、 **.OR.**、 **.XOR.**、 **.EQV.**、 **.NEQV.**)の場合、整数タイプのオペランドと論理タイプのオペランド間、または 2 つの整数タイプのオペランド間の論理演算の結果は、整数タイプになります。 結果の kind 型付きパラメーターは、2 つのオペランドのうちの長い方の長さになります。 2 つのオペランドのパラメーターが同じである場合、結果の kind パラメーターは、その kind パラメーターになります。

- **-qlog4** コンパイラー・オプションを使用し、デフォルトの整数サイズが **INTEGER(4)** の場合は、論理演算の論理結果は、上記の表に示す **LOGICAL(n)** でなく、タイプ **LOGICAL(4)** になります。 **-qlog4** オプションを指定し、デフォルトの整数サイズが **INTEGER(4)** でない場合は、その結果は上記の表に指定したものと同じになります。
- **-qctyplss** コンパイラー・オプションを指定すると、XL Fortran は、文字定数式をホレリス定数として扱います。オペランドのいずれか一方または両方が文字定数式である場合は、その結果のデータ型と長さは、文字定数式がホレリス定数である場合と同じになります。結果のデータ型と長さについては、前記の表の『タイプなし』の行を参照してください。

コンパイラー・オプションの内容については、「ユーザーズ・ガイド」の『*XL Fortran コンパイラー・オプションに関する参照事項*』を参照してください。

IBM 拡張 の終り

BYTE データ・オブジェクトの使用法

IBM 拡張

BYTE タイプのデータ・オブジェクトは、 **LOGICAL(1)**、**CHARACTER(1)**、または **INTEGER(1)** データ・オブジェクトが使用できるところであればどこでも使用することができます。

BYTE データ・オブジェクトのデータ型は、それを使用するコンテキストによって決まります。 XL Fortran は、使用前には変換を行いません。たとえば、名前付き定数のタイプは、最初に割り当てられた値によってではなく、使用することによって決まります。

- **BYTE** データ・オブジェクトの 2 進の算術演算子、論理演算子、関係演算子のオペランドとして使用すると、データ・オブジェクトのデータ型は次のようになります。
  - その他のオペランドが算術、**BYTE**、またはタイプなし定数の場合、**INTEGER(1)**
  - その他のオペランドが論理定数の場合、**LOGICAL(1)**
  - その他のオペランドが文字定数の場合、**CHARACTER(1)**
- **BYTE** データ・オブジェクトを連結演算子のオペランドとして使用すると、データ・オブジェクトのデータ型は、**CHARACTER(1)** になります。
- **BYTE** データ・オブジェクトを明示インターフェースを使用するプロシージャーに対する実引き数として使用すると、データ・オブジェクトのデータ型は対応する仮引き数のタイプになります。
  - **INTEGER(1)** 仮引き数の場合、**INTEGER(1)**
  - **LOGICAL(1)** 仮引き数の場合、**LOGICAL(1)**
  - **CHARACTER(1)** 仮引き数の場合、**CHARACTER(1)**
- **BYTE** データ・オブジェクトを、暗黙インターフェースを使用する外部サブプログラムに対する実引き数 (参照によって渡されている) として使用すると、そのデータ・オブジェクトは、1 バイトの長さになり、データ型は想定されません。
- **BYTE** データ・オブジェクトを値 (**%VAL**) によって渡される実引き数として使用すると、データ・オブジェクトのデータ型は、**INTEGER(1)** になります。
- **BYTE** データ・オブジェクトを、算術、論理、または文字のいずれかの特定のデータ型を必要とするコンテキストの中で使用すると、そのデータ・オブジェクトのデータ型は、それぞれ **INTEGER(1)**、**LOGICAL(1)**、または **CHARACTER(1)** になります。
- **BYTE** タイプのポインターを、文字タイプのターゲットに関連させることはできません。また、文字タイプのポインターを **BYTE** タイプのターゲットに関連させることもできません。
- **BYTE** データ・オブジェクトをその他のコンテキストの中で使用する場合、データ・オブジェクトのデータ型は、**INTEGER(1)** になります。

IBM 拡張 の終り

組み込み割り当て

割り当てステートメントは、式の計算結果に基づいて、変数を定義または再定義する実行可能ステートメントです。

定義済みの割り当ては、組み込みではなく、サブルーチンおよびインターフェース・ブロックによって定義されます。 180 ページの『定義済み割り当て』を参照してください。

組み込み割り当ての一般的な形式は、次のとおりです。

```
►►—variable— = —expression————————————————————————————————————►◄
```

variable と *expression* の形状は、整合性がとれていなければなりません。 *expression* が配列であれば、*variable* も配列でなければなりません (105 ページの『配列にかかわる式』を参照してください)。 *expression* がスカラーで *variable* が配列の場合、その *expression* は、すべての配列エレメントが *expression* のスカラー値と同じ値を持つ *variable* と同じ形状の配列として扱われます。 *variable* は「多対 1」配列セクションにしてはなりません (詳細は 99 ページの『ベクトル添え字』を参照してください)。また、*variable* も *expression* も、想定サイズ配列にはできません。 *variable* と *expression* のタイプは、次に示すように整合性がとれていなければなりません。

variable **のタイプ**	*expression* **のタイプ**
数値	数値
論理	論理
文字	文字
派生型	派生型 (*variable* と同様)

数値割り当てステートメントでは、*variable* および *expression* で、異なる数値タイプと異なる kind 型付きパラメーターを指定できます。論理割り当てステートメントの場合、kind 型付きパラメーターは異なってもかまいません。文字割り当てステートメントの場合、length 型付きパラメーターは異なってもかまいません。

文字変数が文字式よりも長い場合、その文字式は、文字変数と長さが等しくなるまで、ブランクによって右側方向に拡張されます。文字変数が文字式より短い場合、その文字式は、文字変数と長さが文字変数の長さに位置するように、文字式の右側が切り捨てられます。

variable がポインターである場合、その *variable* は、*expression* と整合性がとれたタイプ、型付きパラメーター、および形状を持つ定義可能なターゲットに関連させなければなりません。そこで、*expression* の値は、*variable* に関連したターゲットに割り当てられます。

variable および *expression* の両方に、*variable* のどの部分に対する参照も含めることができます。

割り当てステートメントによって、*expression* および割り当て前の *variable* 内のすべての式の計算、必要に応じて *variable* のタイプや型付きパラメーターに対する *expression* の変換、および結果値による *variable* の定義が行われます。長さがゼロの文字オブジェクトまたはサイズがゼロの配列の場合、*variable* に対して値は割り当てられません。

派生型のオブジェクトに対してアクセス可能な定義済み割り当てがない場合、その派生型割り当てステートメントは、組み込み割り当てステートメントになります。派生型の式は、変数と同じ派生型でなければなりません。 (2 つの構造体が同じ派生型の構造体であるかどうかを判別する規則については、 47 ページの『派生型のタイプの決め方』を参照してください。) 式の各コンポーネントが対応する変数のコンポーネントに割り当てられているかのように、割り当て (またはポインターの割り当て) が行われます。ポインターの割り当ては、ポインターのコンポーネントに対して実行され、組み込み割り当ては、ポインター以外の 割り当て可能でないコンポーネントに対して実行されます。割り当て可能なコンポーネントの場合は、以下の操作の順序が適用されます。

1. *variable* のコンポーネントが現在割り振り済みの場合は、割り振りが解除されます。
2. *expression* のコンポーネントが現在割り振り済みの場合は、対応する *variable* のコンポーネントが、 *expression* のコンポーネントと同じタイプおよび型付きパラメーターを使用して割り振られます。それが配列である場合には、同じ境界で割り振られます。

   それから、*expression* のコンポーネントの値が、組み込み割り当てを使用して、対応する *variable* のコンポーネントに割り当てられます。

variable がサブオブジェクトである場合、割り当ては、定義状況やオブジェクトの他の部分の値に影響しません。

算術変換

数値組み込み割り当ての場合、*expression* の値は、以下の表に指定したように、*variable* のタイプおよび kind 型付きパラメーターに変換されることがあります。

variable **のタイプ**	**割り当てられる値**
整数	INT(*expression*,KIND=KIND(*variable*))
実数	REAL(*expression*,KIND=KIND(*variable*))
複素数	CMPLX(*expression*,KIND=KIND(*variable*))

IBM 拡張

注: データ・オブジェクト **INTEGER(1)**、**INTEGER(2)**、および **INTEGER(4)** の整数演算は、式の計算に、**INTEGER(4)** 算術タイプを使って行われます。 **INTEGER(1)** または **INTEGER(2)** が必要とするコンテキスト中で中間結果が使用される場合は、必要に応じて変換されます。 **INTEGER(8)** データ項目の整数演算は、**INTEGER(8)** 算術を使用して実行されます。

IBM 拡張 の終り

文字割り当て

文字変数を定義するのに必要となる数の文字式だけが計算されます。たとえば、次のようになります。

```
CHARACTER SCOTT*4, DICK*8
SCOTT = DICK
```

SCOTT に DICK を割り当てるには、事前にサブストリング DICK(1:4) を定義しておく必要のみあります。 DICK (DICK(5:8)) の残りの部分を事前に定義しておく必要はありません。

BYTE 割り当て

IBM 拡張

expression が算術式である場合は、算術割り当てを使用します。同様に、*expression* が文字タイプである場合は文字割り当てを使用し、*expression* が論理タイプである場合は論理割り当てを使用します。式の右辺が **BYTE** タイプの場合は、算術割り当てを使用します。

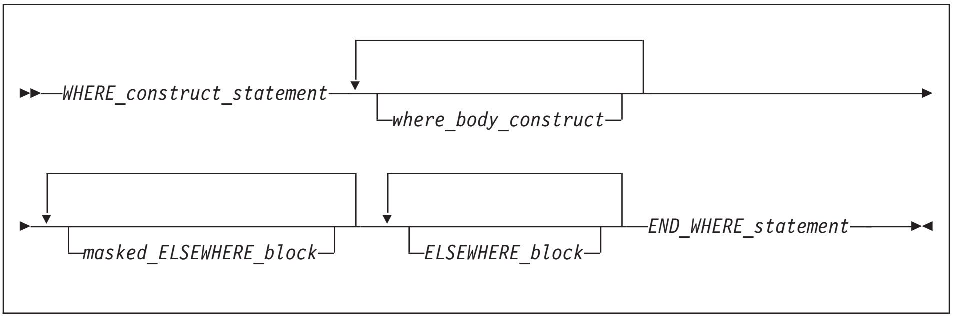
IBM 拡張 の終り

組み込み割り当ての例:

```
INTEGER I(10)
LOGICAL INSIDE
REAL R,RMIN,RMAX
REAL :: A=2.3,B=4.5,C=6.7
TYPE PERSON
   INTEGER(4) P_AGE
   CHARACTER(20) P_NAME
END TYPE
TYPE (PERSON) EMP1, EMP2
CHARACTER(10) :: CH = 'ABCDEFGHIJ'

I = 5                      ! All elements of I assigned value of 5

RMIN = 28.5 ; RMAX = 29.5
R = (-B + SQRT(B**2 - 4.0*A*C))/(2.0*A)
INSIDE = (R .GE. RMIN) .AND. (R .LE. RMAX)

CH(2:4) = CH(3:5)                 ! CH is now 'ACDEEFGHIJ'

EMP1 = PERSON(45, 'Frank Jones')
EMP2 = EMP1

! EMP2%P_AGE is assigned EMP1%P_AGE using arithmetic assignment
! EMP2%P_NAME is assigned EMP1%P_NAME using character assignment

END
```

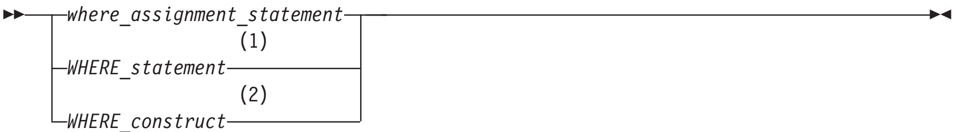
WHERE 構造体

WHERE 構造体は、配列式および配列割り当ての計算をマスクします。これは論理配列式の値に応じて実行されます。



WHERE_construct_statement
構文の詳細については、 497 ページの『WHERE』を参照してください。

where_body_construct



注:

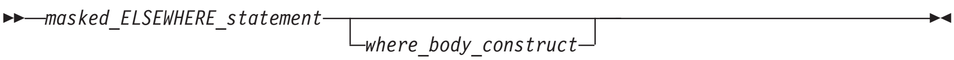
1 Fortran 95 変数

2 Fortran 95 変数

where_assignment_statement
assignment_statement の 1 つです。

Fortran 95

masked_ELSEWHERE_block



masked_ELSEWHERE_statement
mask_expr を指定する **ELSEWHERE** ステートメントです。 構文の詳細については、 348 ページの『ELSEWHERE』を参照してください。

Fortran 95 の終り

ELSEWHERE_block

```
►►—ELSEWHERE_statement—┬──────────────────────┬——————————————►◄
                       └where_body_construct—┘
```

ELSEWHERE_statement
: *mask_expr* を指定しない **ELSEWHERE** ステートメントです。 構文の詳細については、 348 ページの『ELSEWHERE』を参照してください。

END_WHERE_statement
: 構文の詳細については、 351 ページの『END (構造体)』を参照してください。

規則:

- *mask_expr* は、論理配列式です。
- *where_assignment_statement* では、 *mask_expr* および定義される変数 *variable* は、同じ形状の配列でなければなりません。
- *where_body_construct* の一部であるステートメントは、分岐ターゲット・ステートメントにすることはできません。さらに、**ELSEWHERE**、マスクされた **ELSEWHERE**、および **END WHERE** ステートメントは、分岐ターゲット・ステートメントにすることはできません。

Fortran 95

- 定義済み割り当てである *where_assignment_statement* は、エレメント型の定義済み割り当てでなければなりません。
- **WHERE** 構造体ステートメント上の *mask_expr* およびすべての対応するマスクされた **ELSEWHERE** ステートメントは同じ形状でなければなりません。ネストした **WHERE** ステートメント上の *mask_expr* またはネストした **WHERE** 構造体ステートメントは、そのネストを含む構造体の **WHERE** 構造体ステートメント上の *mask_expr* と同じ形状でなければなりません。
- 構造体名を **WHERE** 構造体ステートメントに指定する場合、対応する **END WHERE** ステートメントにも指定しなければなりません。 構造体名は、マスクされた **ELSEWHERE** および **WHERE** 構造体内の **ELSEWHERE** ステートメントでは任意指定です。

Fortran 95 の終り

マスクされた配列割り当ての解釈

マスクされた配列割り当ての解釈方法を理解するには、*制御マスク* (m_c) および*保留制御マスク* (m_p) の概念について理解する必要があります。

- m_c は、論理タイプの配列で、その値によって *where_assignment_statement* のどの配列のエレメントを定義するかが決定されます。この値は、以下のどれを実行するかによって決定されます。
  - **WHERE** ステートメント
  - **WHERE** 構造体ステートメント
  - **ELSEWHERE** ステートメント
  - F95 マスクされた **ELSEWHERE** ステートメント F95
  - **END WHERE** ステートメント

  m_c の値は累積的です。つまり、コンパイラーは前後の **WHERE** ステートメントのマスク式および現在のマスク式を使ってその値を決定します。 *mask_expr* 内のエンティティーの値に対して後から変更が加えられても、m_c の値には影響はありません。コンパイラーは、*mask_expr* を、**WHERE** ステートメント、**WHERE** 構造体ステートメント、または F95 マスクされた **ELSEWHERE** ステートメント F95 ごとに 1 度だけ評価します。
- m_p は論理配列で、現在の **WHERE** ステートメント、**WHERE** 構造体ステートメント、 F95 またはマスクされた **ELSEWHERE** ステートメント F95 によって定義されていない配列エレメント上の同じネスト・レベルで、次のマスクされた割り当てステートメントに情報を提供します。

以下では、コンパイラーが **WHERE**、**WHERE** 構造体、 F95 マスクされた **ELSEWHERE** F95 、 **ELSEWHERE**、または **END WHERE** ステートメント内のステートメントを解釈する方法を説明しています。ここでは m_c や m_p、およびこれ以外のステートメントの動作に与える影響について、発生の順番に説明しています。

- **WHERE** ステートメント

  Fortran 95

  - **WHERE** ステートメントが **WHERE** 構造体内でネストしている場合、以下のことが起きます。
    1. m_c は、m_c **.AND.** *mask_expr* になります。
    2. コンパイラーが **WHERE** ステートメントを実行した後、m_c は **WHERE** ステートメントの実行前に持っていた値を持ちます。

  Fortran 95 の終り

  - そうでない場合は、m_c は *mask_expr* になります。

- **WHERE** 構造体

Fortran 95

– **WHERE** 構造体が別の **WHERE** 構造体内でネストしている場合、以下のことが起きます。

1. m_p は、m_c **.AND.** (**.NOT.** *mask_expr*) になります。
2. m_c は、m_c **.AND.** *mask_expr* になります。

Fortran 95 の終り

– 上記のようにならない場合、以下のようになります。

1. コンパイラーは *mask_expr* を評価し、m_c にその *mask_expr* の値を割り当てます。
2. m_p は **.NOT.** *mask_expr* になります。

Fortran 95

- マスクされた **ELSEWHERE** ステートメント

  以下のことが起きます。

  1. m_c は m_p になります。
  2. m_p は、m_c **.AND.** (**.NOT.** *mask_expr*) になります。
  3. m_c は、m_c **.AND.** *mask_expr* になります。

Fortran 95 の終り

- **ELSEWHERE** ステートメント

  以下のことが起きます。

  1. m_c は m_p になります。新しい m_p の値は設定されません。
- **END WHERE** ステートメント

  コンパイラーが **END WHERE** ステートメントを実行した後、m_c と m_p は、対応する **WHERE** 構造体ステートメントの実行前に持っていた値を持ちます。
- *where_assignment_statement*

  コンパイラーは、m_c の真の値に対応する *expr* の値を、対応する *variable* のエレメントに割り当てます。

非エレメント型関数参照が *where_assignment_statement* 内の *expr* または *variable*、または *mask_expr* にある場合、コンパイラーはマスクされた制御なしで関数を評価します。つまり、完全に関数の引き数式のすべてを評価し、それによって完全に関数を評価しま

す。結果が配列で、参照が非エレメント型関数の引き数リストの範囲内にない場合、*expr*、*variable*、または *mask_expr* の計算に使用するために m_c 内の真の値に対応するエレメントが選択されます。

エレメント型の組み込み演算または関数参照が *where_assignment_statement* または *mask_expr* の *expr* または *variable* で発生し、非エレメント型関数参照の引き数リストの範囲内にない場合、m_c 内の真の値に対応するエレメントに対してのみ、演算が実行されるか、あるいは関数が計算されます。

配列コンストラクターが *where_assignment_statement* または *mask_expr* にある場合、コンパイラーは配列コンストラクターを、マスクされた制御なしで評価し、それから *where_assignment_statement* を実行するか、または *mask_expr* を評価します。

WHERE ステートメントの *mask_expr* 内の関数参照を実行することによって、*where_assignment_statement* 内のエンティティーに影響を与えることができます。 **END WHERE** を実行しても、影響はありません。

以下の例は、制御マスクが更新される方法を示しています。この例で、*mask1*、*mask2*、*mask3*、および *mask4* は適合論理配列で、m_c は制御マスクで、m_p は保留制御マスクです。コンパイラーはそれぞれのマスク式を一度評価します。

サンプル・コード (注釈内にステートメント番号を示しています)

```
WHERE (mask1)        ! W1 *
  WHERE (mask2)      ! W2 *
  ...                ! W3 *
  ELSEWHERE (mask3)  ! W4 *
  ...                ! W5 *
  END WHERE          ! W6 *
ELSEWHERE (mask4)    ! W7 *
...                  ! W8 *
ELSEWHERE            ! W9
...                  ! W10
END WHERE            ! W11
```

注: * Fortran 95

以下に示すとおり、コンパイラーは制御および保留制御マスクを、それぞれのステートメントを実行するごとに設定します。

Fortran 95

Statement W1

m_c = mask1

m_p = .NOT. mask1

Statement W2

m_p = mask1 .AND. (.NOT. mask2)

m_c = mask1 .AND. mask2

```
Statement W4
    m_c = mask1 .AND. (.NOT. mask2)
    m_p = mask1 .AND. (.NOT. mask2)
.AND. (.NOT. mask3)
    m_c = mask1 .AND. (.NOT. mask2)
.AND. mask3
Statement W6
    m_c = mask1
    m_p = .NOT. mask1
```

Fortran 95 の終り

```
Statement W7
    m_c = .NOT. mask1
    m_p = (.NOT. mask1) .AND. (.NOT.
mask4)
    m_c = (.NOT. mask1) .AND. mask4
Statement W9
    m_c = (.NOT. mask1) .AND. (.NOT.
mask4)
Statement W11
    m_c = 0
    m_p = 0
```

コンパイラーは、ステートメント *W2*、*W4*、*W7*、および *W9* によって設定された制御マスクの値を、それぞれ *where_assignment_statement* の *W3*、*W5*、*W8*、および *W10* を実行するときに使用します。

マイグレーションのためのヒント:

配列の論理計算を単純化します。

FORTRAN 77 ソース

```
INTEGER A(10,10),B(10,10)
   ⋮
DO I=1,10
  DO J=1,10
    IF (A(I,J).LT.B(I,J)) A(I,J)=B(I,J)
  END DO
END DO
END
```

Fortran 90 または Fortran 95 ソース:

```
INTEGER A(10,10),B(10,10)
   ⋮
WHERE (A.LT.B) A=B
END
```

WHERE 構造体の例

```
REAL, DIMENSION(10) :: A,B,C,D
WHERE (A>0.0)
  A = LOG(A)          ! Only the positive elements of A
                      !   are used in the LOG calculation.
  B = A               ! The mask uses the original array A
                      !   instead of the new array A.
  C = A / SUM(LOG(A)) ! A is evaluated by LOG, but
                      !   the resulting array is an
                      !   argument to a non-elemental
                      !   function. All elements in A will
                      !   be used in evaluating SUM.
END WHERE

WHERE (D>0.0)
  C = CSHIFT(A, 1)    ! CSHIFT applies to all elements in array A,
                      ! and the array element values of D determine
                      ! which CSHIFT expression determines the
                      ! corresponding element values of C.
ELSEWHERE
  C = CSHIFT(A, 2)
END WHERE
END
```

Fortran 95

以下の例は、**WHERE** 構造体ステートメント内およびマスクされた **ELSEWHERE** *mask_expr* 内の配列コンストラクターを示しています。

```
CALL SUB((/ 0, -4, 3, 6, 11, -2, 7, 14 /))

CONTAINS
  SUBROUTINE SUB(ARR)
  INTEGER ARR(:)
  INTEGER N

  N = SIZE(ARR)

  ! Data in array ARR at this point:
  !
  ! A = | 0 -4 3 6 11 -2 7 14 |

  WHERE (ARR < 0)
    ARR = 0
  ELSEWHERE (ARR < ARR((/(N-I, I=0, N-1)/)))
    ARR = 2
  END WHERE

  ! Data in array ARR at this point:
  !
  ! A = | 2 0 3 2 11 0 7 14 |

  END SUBROUTINE
END
```

以下の例は、ネストされた **WHERE** 構造体ステートメントおよびマスクされた **ELSEWHERE** ステートメントに *where_construct_name* を指定したものを示しています。

```
INTEGER :: A(10, 10), B(10, 10)
...
OUTERWHERE: WHERE (A < 10)
  INNERWHERE: WHERE (A < 0)
    B = 0
  ELSEWHERE (A < 5) INNERWHERE
    B = 5
  ELSEWHERE INNERWHERE
    B = 10
  END WHERE INNERWHERE
ELSEWHERE OUTERWHERE
  B = A
END WHERE OUTERWHERE
...
```

Fortran 95 の終り

FORALL 構造体

Fortran 95

FORALL 構造体は、サブオブジェクトのグループへの割り当て、特に配列エレメントへの割り当てを実行します。

WHERE 構造体とは異なり、**FORALL** は、配列エレメント、配列セクションおよびサブストリングの割り当てを実行します。また、**FORALL** 構造体の中のそれぞれの割り当ては、直前の割り当てと整合していなくても構いません。 **FORALL** 構造体には、ネストされた **FORALL** ステートメント、**FORALL** 構造体、 **WHERE** ステートメントおよび **WHERE** 構造体を含むことができます。

Fortran 95 の終り

IBM 拡張

INDEPENDENT ディレクティブは、**FORALL** ステートメントまたは構造体の各演算をどの順序で行ってもプログラムのセマンティクスに影響しないですむようにします。 **INDEPENDENT** ディレクティブの詳細については、 519 ページの『INDEPENDENT』を参照してください。

IBM 拡張 の終り

Fortran 95

```
►►──FORALL_construct_statement─────────────────────────────────────►◄

►►──forall_body────────────────────────────────────────────────────►◄

►►──END_FORALL_statement───────────────────────────────────────────►◄
```

FORALL_construct_statement
構文の詳細については、 371 ページの『FORALL (構造体)』を参照してください。

END_FORALL_statement
構文の詳細については、 351 ページの『END (構造体)』を参照してください。

forall_body
次のステートメントまたは構造体の 1 つ以上です。
forall_assignment

WHERE ステートメント (497 ページの『WHERE』を参照)
WHERE 構造体 (131 ページの『WHERE 構造体』を参照)
FORALL ステートメント (367 ページの『FORALL』を参照)
FORALL 構造体

forall_assignment
assignment_statement または *pointer_assignment_statement* のどちらかです。

forall_body で参照されるプロシージャーはすべて (定義済み操作または定義済み割り当てによって参照されるものを含む)、純粋でなければなりません。

FORALL ステートメントまたは構造体が、**FORALL** 構造体の中でネストされている場合、内側の **FORALL** ステートメントまたは構造体は、外側の **FORALL** 構造体で使用されているどの *index_name* も再定義することはできません。

同じステートメントの中で、複数回、アトミック・オブジェクトを割り当てたり、関連付け状況を変更したりすることはできませんが、同じ **FORALL** 構造体の中にある別の割り当てステートメントは、アトミック・オブジェクトを再定義したり再び関連させたりすることができます。また、**WHERE** 構造体の中のそれぞれの **WHERE** ステートメントおよび割り当てステートメントは、次の制約事項を守らなければなりません。

FORALL_construct_name を指定する場合、**FORALL** ステートメントおよび **END FORALL** ステートメントの両方で指定しなければなりません。 **END FORALL** ステートメントまたは **FORALL** 構造体の中のどのステートメントも、分岐ターゲット・ステートメントにできません。

Fortran 95 の終り

FORALL 構造体の解釈

Fortran 95

1. **FORALL** 構造体ステートメントの中の各 *forall_triplet_spec* の *subscript* および *stride* 式は、順序に関係なく評価されます。可能な *index_name* 値のすべての組が、組み合わせの集合を形成します。たとえば、次のようなステートメントを想定します。

   ```
   FORALL (I=1:3,J=4:5)
   ```

   I および J の組み合わせの集合は次のとおりです。

   ```
   {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}
   ```

   -1 および **-qnozerosize** コンパイラー・オプションは、このステップに影響を与えません。

2. 組み合わせの集合の *scalar_mask_expr* は順序に関係なく評価され (**FORALL** 構造体ステートメントにある)、アクティブな組み合わせの集合が作成されます (.TRUE. と

評価されたもの)。たとえば、(`I+J.NE.6`) が上記の集合に適用された場合、アクティブな組み合わせの集合は次のようになります。

```
{(1,4),(2,5),(3,4),(3,5)}
```

3. 出現順で、それぞれの *forall_body* ステートメントまたは *forall_body* 構造体を実行します。アクティブな組み合わせの集合に対して、それぞれのステートメントまたは構造体は、次のように完全に実行されます。

assignment_statement

すべてのアクティブな *index_name* 値の組み合わせについて、順序に関係なく、右側の *expression* のすべての値、および左側の *variable* のすべての添え字、ストライド、およびサブストリング境界を評価します。

すべてのアクティブな *index_name* 値について、順序に関係なく、計算された *expression* の値を対応する *variable* エンティティーに割り当てます。

```
INTEGER, DIMENSION(50) :: A,B,C
INTEGER :: X,I=2,J=49
FORALL (X=I:J)
  A(X)=B(X)+C(X)
  C(X)=B(X)-A(X) ! All these assignments are performed after the
                 ! assignments in the preceding statement
END FORALL
END
```

pointer_assignment_statement

何がポインター割り当てのターゲットになるのかを順序に関係なく評価し、すべてのアクティブな *index_name* 値の組み合わせについて、ポインターのすべての添え字、ストライド、およびサブストリング境界を評価します。ターゲットがポインターではない場合、ターゲットの判別には、その値の評価は含まれません。ポインター割り当ては、右側の値を判別する必要 はありません。

すべてのアクティブな *index_name* 値の組み合わせに対して、順序に関係なく、すべてのターゲットを、対応するポインター・エンティティーに関連させます。

WHERE ステートメントまたは構造体

WHERE ステートメント、**WHERE** 構造体ステートメント、**ELSEWHERE** ステートメント、またはマスクされた **ELSEWHERE** ステートメント (それぞれアクティブな *index_name* 値の組み合わせ) ごとに、順序に関係なく、制御マスクおよび保留制御マスクを評価し、そのステートメントにとってよりよいアクティブな組み合わせを作ります。これについては 133 ページの『マスクされた配列割り当ての解釈』で説明しています。それぞれのアクティブな組み合わせに対し、コンパイラーは **WHERE** ステートメント、**WHERE** 構造体ステートメント、またはマスクされた **ELSEWHERE** ステートメントの割り当てを実行し、そのアクティブな組み

合わせで真である制御マスクの値を割り当てます。 前に説明したとおり、コンパイラーは、**WHERE** 構造体内のそれぞれのステートメントを順に実行します。

```
INTEGER I(100,10), J(100), X
FORALL (X=1:100, J(X)>0)
  WHERE (I(X,:)<0)
    I(X,:)=0  ! Assigns 0 to an element of I along row X
              ! only if element value is less than 0 and value
              ! of element in corresponding column of J is
  ELSEWHERE   ! greater than 0.
    I(X,:)=1
  END WHERE
END FORALL
END
```

FORALL ステートメントまたは構造体

外側の **FORALL** ステートメントまたは構造体にあるアクティブな組み合わせについて、順序に関係なく、*forall_triplet_spec_list* の中の *subscript* 式および *stride* 式を評価します。有効な組み合わせは、内側と外側の **FORALL** 構造体の組み合わせの集合のカルテシアン積です。 *scalar_mask_expr* は、内側の **FORALL** 構造体のアクティブな組み合わせを判別します。これらのアクティブな組み合わせについて、ステートメントと構造体が実行されます。

```
! Same as FORALL (I=1:100,J=1:100,I.NE.J) A(I,J)=A(J,I)

INTEGER A(100,100)
OUTER: FORALL (I=1:100)
  INNER: FORALL (J=1:100,I.NE.J)
    A(I,J)=A(J,I)
  END FORALL INNER
END FORALL OUTER
END
```

Fortran 95 の終り

ポインターの割り当て

ポインターの割り当てステートメントによって、ポインターがターゲットと関連させられます。また、ポインター割り当てステートメントによって、ポインターの関連付け状況は、関連のなくなった状態または未定義の状態にもなります。

```
►►—pointer_object— => —target————————————————————————————————►◄
```

target 変数または式です。 target は *pointer_object* と同じタイプ、型付きパラメーター、およびランクを持たなければなりません。

pointer_object は、**POINTER** 属性を持たなければなりません。

式であるターゲットによって、**POINTER** 属性を持つ値が得られなければなりません。変数であるターゲットは、**TARGET** 属性 (またはそのようなオブジェクトの対象となるもの) あるいは、変数であるターゲットは、**POINTER** 属性を持たなければなりません。ターゲットは、ベクトル添え字を持つ配列セクションであってはなりません。また、整数の想定サイズ配列であってはなりません。

関連解除された配列ポインターのターゲットのサイズ、境界、および形状は未定義です。こうした配列の一部を定義したり、参照したりはできません。ただし、配列を、関連付け状況、引き数の有無、タイプまたは型付きパラメーターの特性を参照する組み込み照会関数の引き数にすることはできます。

IBM 拡張

バイト・タイプのポインターは、バイト・タイプ、**INTEGER(1)** タイプ、 **LOGICAL(1)** タイプのターゲットにのみ関連させることができます。

IBM 拡張 の終り

pointer_object とターゲット間での以前の関連付けはすべて無効にされます。 *target* がポインターでない場合、*pointer_object* は、*target* と関連付けられます。 *target* 自体がポインターに関連させられる場合、*pointer_object* は *target* のターゲットに関連させられます。 *target* が関連解除または未定義の状態の関連付け状況を持つポインターの場合、*pointer_object* は同様の状態となります。ポインター割り当ての *target* が割り振り可能なオブジェクトである場合は、割り振り済みでなければなりません。

ポインター構造体コンポーネントのポインター割り当ては、派生型の組み込み割り当てステートメントまたは定義済み割り当てステートメントの実行によって発生します。

配列ポインターの割り当て時、各次元の下限は、ターゲットの対応する次元に適用される **LBOUND** 組み込み関数の結果となります。全体配列でも構造体コンポーネントでもない配列セクションや配列式の場合、下限は 1 になります。各次元の上限は、ターゲットの対応する次元に適用される **UBOUND** 組み込み関数の結果となります。

関連情報:

ポインターをターゲットに関連させる代替形式については、 290 ページの『ALLOCATE』を参照してください。
プロシージャー参照の中でのポインターの使用法の詳細については、 204 ページの『仮引き数としてのポインター』を参照してください。

ポインター割り当ての例

```
TYPE T
  INTEGER, POINTER :: COMP_PTR
ENDTYPE T
TYPE(T) T_VAR
INTEGER, POINTER :: P,Q,R
INTEGER, POINTER :: ARR(:)
BYTE, POINTER :: BYTE_PTR
LOGICAL(1), POINTER :: LOG_PTR
INTEGER, TARGET :: MYVAR
INTEGER, TARGET :: DARG(1:5)
P => MYVAR                ! P points to MYVAR
Q => P                    ! Q points to MYVAR
NULLIFY (R)               ! R is disassociated
Q => R                    ! Q is disassociated
T_VAR = T(P)              ! T_VAR%COMP_PTR points to MYVAR
ARR => DARG(1:3)
BYTE_PTR => LOG_PTR
END
```

整数ポインターの割り当て

IBM 拡張

整数ポインター変数は、次のことができます。

- 整数式で使用できます。
- 絶対アドレスとしての割り当て値となります。
- **LOC** 組み込み関数を使用して、変数のアドレスを割り当てることができます。(派生型のオブジェクトおよび構造体コンポーネントは、**LOC** 組み込み関数とともに使用する場合、順序派生型でなければなりません。)

XL Fortran コンパイラーは、割り当てステートメント内の整数ポインターに対して、 1 バイトの算術を使用することに注意してください。

整数ポインターの割り当ての例

```
INTEGER INT_TEMPLATE
POINTER (P,INT_TEMPLATE)
INTEGER MY_ARRAY(10)
DATA MY_ARRAY/1,2,3,4,5,6,7,8,9,10/
INTEGER, PARAMETER :: WORDSIZE=4

P = LOC(MY_ARRAY)
PRINT *, INT_TEMPLATE          ! Prints '1'
P = P + 4;                     ! Add 4 to reach next element
                               !    because arithmetic is byte-based
PRINT *, INT_TEMPLATE          ! Prints '2'

P = LOC(MY_ARRAY)
DO I = 1,10
```

```
  PRINT *,INT_TEMPLATE
  P = P + WORDSIZE             ! Parameterized arithmetic is suggested
END DO
END
```

IBM 拡張 の終り

第 6 章 制御構造

この章では、以下の項目について説明します。

- 『ステートメント・ブロック』
- 『IF 構造体』
- 149 ページの『CASE 構造体』
- 152 ページの『DO 構造体』
- 157 ページの『DO WHILE 構造体』
- 157 ページの『分岐』

24 ページの『ステートメントおよび実行の順序』で定義したように、通常の実行順序を変更できるステートメント・ブロックおよび他の実行可能ステートメントを含む構造体によって実行順序を制御することができます。この章で取り扱う構造体では、構造体ステートメントの詳細な構文は説明していません。『ステートメント』の項を参照してください。

構造体が別の構造体の中に組み込まれている場合、一方の構造体がもう一方の構造体に完全に収まるような形 (ネスト) になっていなければなりません。ステートメントで構造体名を指定すると、そのステートメントは指定した構造体に属します。指定しないと、ステートメントは一番内側の構造体に属します。

ステートメント・ブロック

ステートメント・ブロック は、別の実行可能構造体に組み込まれて 1 つの単位として扱われる、ゼロ個以上の実行可能ステートメント、実行可能構造体、**FORMAT** ステートメント、 **DATA** ステートメントにより構成されます。

実行可能プログラム内では、ステートメント・ブロックの外側から内側に制御を移すことはできません。ステートメント・ブロック内で、または、ステートメント・ブロックの内側から外側に制御を移すことは可能です。たとえば、ステートメント・ブロック内では、ステートメント・ラベル付きのステートメントおよびそのラベルを使った **GO TO** ステートメントを使用することができます。

IF 構造体

IF 構造体では、実行対象として、1 つのステートメント・ブロックのみを選択します。


```
►►─Block_IF_statement──────────────────────────────────────►◄

►►─statement_block─────────────────────────────────────────►◄

►►──┬──────────────────┬───────────────────────────────────►◄
    │ ┌──────────────┐ │
    │ ▼              │ │
    └──ELSE_IF_block─┴─┘

►►──┬────────────┬─────────────────────────────────────────►◄
    └─ELSE_block─┘

►►─END_IF_statement────────────────────────────────────────►◄
```

Block_IF_statement
: 構文の詳細については、 387 ページの『IF (ブロック)』を参照してください。

END_IF_statement
: 構文の詳細については、 351 ページの『END (構造体)』を参照してください。

ELSE_IF_block

```
►►─ELSE_IF_statement───────────────────────────────────────►◄

►►─statement_block─────────────────────────────────────────►◄
```

ELSE_IF_statement
: 構文の詳細については、 347 ページの『ELSE IF』を参照してください。

ELSE_block

```
►►─ELSE_statement──────────────────────────────────────────►◄

►►─statement_block─────────────────────────────────────────►◄
```

ELSE_statement
: 構文の詳細については、 346 ページの『ELSE』を参照してください。

IF 構造体 (つまり、ブロック **IF** および **ELSE IF** ステートメント) 内のスカラー論理式は、真の値、**ELSE** ステートメント、または **END IF** ステートメントが検出されるまで、指定した順に計算されます。

- 真の値または **ELSE** ステートメントが検出されると、直後のステートメント・ブロックが実行され、**IF** 構造体が完了します。 **IF** 構造体に残っている **ELSE IF** ステートメントまたは **ELSE** ステートメント内のスカラー論理式があっても、それらは計算されません。
- **END IF** ステートメントが検出されると、どのステートメント・ブロックも実行されずに、**IF** 構造体が完了します。

IF 構造体名を指定する場合、その構造体名は **IF** ステートメントおよび **END IF** ステートメントに必ず指定が必要ですが、 **ELSE IF** ステートメントまたは **ELSE** ステートメントへの指定はオプションとなります。

例

```
! Get a record (containing a command) from the terminal

    DO
      WHICHC: IF (CMD .EQ. 'RETRY') THEN       ! named IF construct
           IF (LIMIT .GT. FIVE) THEN           ! nested IF construct
!              Print retry limit exceeded
               CALL STOP
           ELSE
               CALL RETRY
           END IF
      ELSE IF (CMD .EQ. 'STOP') THEN WHICHC    ! ELSE IF blocks
           CALL STOP
      ELSE IF (CMD .EQ. 'ABORT') THEN
           CALL ABORT
      ELSE WHICHC                              ! ELSE block
!          Print unrecognized command
      END IF WHICHC
    END DO
    END
```

CASE 構造体

CASE 構造体は、実行対象となる多くのステートメント・ブロックの中から 1 つを選択するための簡潔な構文を持っています。 各 **CASE** ステートメントのケース・セレクターは、**SELECT CASE** ステートメントのケース式と似ています。

```
>>--SELECT_CASE_statement--------------------------------------><

>>--+-----------------------------+----------------------------><
    |  .---------------------.    |
    |  V                     |    |
    '----CASE_statement_block-+---'

>>--END_SELECT_statement---------------------------------------><
```

SELECT_CASE_statement
: 計算対象のケース式を定義します。構文の詳細については、 465 ページの『SELECT CASE』を参照してください。

END_SELECT_statement
: **CASE** 構造体を終了します。構文の詳細については、 351 ページの『END (構造体)』を参照してください。

CASE_statement_block

```
>>--CASE_statement---------------------------------------------><

>>--statement_block--------------------------------------------><
```

CASE_statement
: 値、値のセット、またはデフォルト・ケースのいずれかで、後続のステートメント・ブロックが実行されるケース・セレクターを定義します。構文の詳細については、 304 ページの『CASE』を参照してください。

構造体内では、各ケース値のタイプは、ケース式のタイプと同じでなければなりません。

CASE 構造体は、次のように実行されます。

1. ケース式が計算されます。結果値は、ケース指標です。
2. ケース指標は、各 **CASE** ステートメントの *case_selector* と比較されます。
3. 一致すると、**CASE** ステートメントに関連したステートメント・ブロックが実行されます。一致しなければ、どのステートメント・ブロックも実行されません。 (304 ページの『CASE』を参照してください。)
4. 構造体の実行が完了すると、制御が **END SELECT** ステートメントの後に移されます。

CASE 構造体には、それぞれに値の範囲を 1 つ指定できるゼロ個以上の **CASE** ステートメントが入ります。ただし、**CASE** ステートメントで指定する値の範囲をオーバーラップさせることはできません。

複数の **CASE** ステートメントのうちの 1 つで、デフォルトの *case_selector* を指定することができます。デフォルトの *CASE_statement_block* は、**CASE** 構造体の始め、構造体の終わり、あるいは他のブロックの間など、構造体内ならどこにあってもかまいません。

構造体名を指定する場合、その構造体名は **SELECT CASE** ステートメントおよび **END SELECT** ステートメントに必ず指定する必要がありますが、**CASE** ステートメントへの指定はオプションです。

CASE 構造体内からは、**END SELECT** ステートメントに対してのみ分岐できます。**CASE** ステートメントは分岐ターゲットにはなれません。

マイグレーションのためのヒント:

IF ブロックの代わりに **CASE** を使用します。

FORTRAN 77 ソース

```
IF (I .EQ.3) THEN
     CALL SUBA()
ELSE IF (I.EQ. 5) THEN
     CALL SUBB()
ELSE IF (I .EQ. 6) THEN
     CALL SUBC()
ELSE
     CALL OTHERSUB()
ENDIF
END
```

Fortran 90 または Fortran 95 ソース

```
SELECTCASE(I)
  CASE(3)
    CALL SUBA()
  CASE(5)
    CALL SUBB()
  CASE(6)
    CALL SUBC()
  CASE DEFAULT
    CALL OTHERSUB()
END SELECT
END
```

例

```
ZERO: SELECT CASE(N)

  CASE DEFAULT ZERO
       OTHER: SELECT CASE(N) ! start of CASE construct OTHER
          CASE(:-1)
             SIGNUM = -1     ! this statement executed when n≤-1
          CASE(1:) OTHER
              SIGNUM = 1
       END SELECT OTHER      ! end of CASE construct OTHER
  CASE (0)
    SIGNUM = 0

END SELECT ZERO
END
```

DO 構造体

DO 構造体は、ステートメント・ブロックの実行の繰り返しを指定します。このように繰り返し実行されるブロックをループ と呼びます。

ループの繰り返し回数は、無限でない限り、**DO** 構造体の実行の開始時に決定されます。

CYCLE ステートメントを使用して特定の繰り返しを短縮したり、**EXIT** ステートメントを使用してループを終了することができます。

```
►►─DO_statement─────────────────────────────────────────────►◄

►►─statement_block──────────────────────────────────────────►◄

►►──┬─END_DO_statement───┬─────────────────────────────────►◄
    └─terminal_statement─┘
```

DO_statement
: 構文の詳細については、 335 ページの『DO』を参照してください。

END_DO_statement
: 構文の詳細については、 351 ページの『END (構造体)』を参照してください。

terminal_statement
: **DO** 構造体を終了するステートメントです。以下の記述を参照してください。

DO ステートメントに **DO** 構造体名を指定する場合、同じ構造体名が指定されている **END DO** ステートメントでその構造体を終了しなければなりません。逆に、**DO** ステ

ートメントに **DO** 構造体名を指定せずに、**END DO** ステートメントで **DO** 構造体を終了させる場合は、**END DO** ステートメントに **DO** 構造体名を指定してはいけません。

終端ステートメント

終端ステートメントは、**DO** ステートメントの後に位置し、実行可能でなければなりません。終端ステートメントとして使用できるステートメントの一覧については、 285 ページの『第 10 章 ステートメントおよび属性』を参照してください。**DO** 構造体の終端ステートメントが論理 **IF** ステートメントの場合、その終端ステートメントには、任意の実行可能ステートメントを含めることができます。ただし、論理 **IF** ステートメントに関する制約が適用されるステートメントは除きます。

DO ステートメント上にステートメント・ラベルを指定した場合は、同じステートメント・ラベルの付いたステートメントで **DO** 構造体を終了させなければなりません。

ラベル付き **DO** ステートメントを同じラベルが付いている **END DO** ステートメントで終了させることはできますが、ラベルなし **END DO** ステートメントで終了させることはできません。 **DO** ステートメントにラベルを指定しない場合は、**END DO** ステートメントで **DO** 構造体を終了させなければなりません。

ネストされたラベル付きの **DO** 構造体および **DO WHILE** 構造体は、終端ステートメントがラベル付きでかつ **END DO** ステートメントでなければ、同じ終端ステートメントを共用することができます。

DO 構造体の範囲

DO 構造体の範囲には、**DO** ステートメント以後、終端ステートメントまでの間にあるすべての実行可能ステートメントが含まれます (終端ステートメントも含まれます)。構造体の範囲に関する規則が適用されるだけでなく、最も内側の共用 **DO** 構造体から共用終端ステートメントへのみ、制御を移すことができます。

アクティブの DO 構造体および非アクティブの DO 構造体

DO 構造体は、アクティブか非アクティブのどちらかになっています。**DO** 構造体は、最初は非アクティブで、**DO** ステートメントが実行されると、アクティブになります。アクティブになった **DO** 構造体が非アクティブになるのは次の場合だけです。

- 繰り返し回数がゼロになった場合
- **DO** 構造体の範囲内で、**RETURN** ステートメントが実行された場合
- 同じ有効範囲単位内にあって、その **DO** 構造体の範囲外にあるステートメントに制御が移された場合
- **DO** 構造体の中から呼び出されたサブルーチンが、選択戻り指定子を使って、その **DO** 構造体の範囲外のステートメントに戻った場合
- **DO** 構造体に属する **EXIT** ステートメントが実行された場合
- **DO** 構造体の範囲内にあるが、外側の **DO** または **DO WHILE** 構造体に属する **EXIT** ステートメントまたは **CYCLE** ステートメントが実行された場合

- **STOP** ステートメントが実行されるか、または何らかの理由でプログラムが停止した場合

DO 構造体が非アクティブになると、**DO** 変数は、割り当てられた最後の値を保持します。

DO ステートメントの実行

無限 **DO** は、無限回数のループを実行します。

ループが無限 **DO** でない場合、**DO** ステートメントは、初期パラメーター、終端パラメーター、およびオプションで増分値を含みます。

1. **DO** ステートメント式 (*a_expr1*、*a_expr2*、および *a_expr3*) を計算して得られた値が、初期パラメーター m_1、終端パラメーター m_2、および増分値 m_3 に設定されます。式を計算する場合、必要に応じて、算術変換の規則にしたがって、**DO** 変数のタイプへの変換が行われます。 (129 ページの『算術変換』を参照してください。) *a_expr3* を指定しないと、m_3 の値は 1 になります。m_3 の値をゼロにしてはなりません。
2. **DO** 変数は、初期パラメーター (m_1) の値によって、定義済み状態になります。
3. 繰り返し回数は、以下の式によって決定されて、設定されます。

   MAX (INT ((m_2 - m_1 + m_3) / m_3), 0)

   次の場合には、繰り返し回数がゼロになることに注意してください。

   $m_1 > m_2$ で $m_3 > 0$, または

   $m_1 < m_2$ で $m_3 < 0$

DO 変数が定義されていないと、繰り返し回数を計算することはできません。この構造体を無限 **DO** 構造体といいます。

IBM 拡張

kind 1、2、または 4 の整変数の場合、繰り返し回数は 2**31 - 1 を超えることはできません。 kind 8 の整変数の場合、 2**63 - 1 を超えることはできません。計算時にオーバーフローまたはアンダーフロー状態になる場合は、回数は未定義になります。

IBM 拡張 の終り

DO ステートメントの実行が完了すると、ループ制御の処理が開始されます。

ループ制御の処理

ループ制御の処理は、**DO** 構造体の範囲を、それ以上実行する必要があるかどうかを判断します。繰り返し回数が調べられ、回数がゼロでなければ、 **DO** 構造体の範囲内の最初のステートメントの実行が開始されます。 繰り返し回数がゼロならば、**DO** 構造体は非アクティブになります。 その結果、当該 **DO** 構造体を共用しているすべての **DO** 構

造体が非アクティブになる場合は、その終端ステートメントの後にある次の実行可能ステートメントが実行されて、通常の実行が続けられます。ただし、終端ステートメントを共用している **DO** 構造体で、アクティブのものがある場合は、最も内側のアクティブな **DO** 構造体の増分値の処理に進みます。

範囲の実行

ステートメント・ブロックの一部を構成しているステートメントは、 **DO** 構造体の範囲内にあります。このステートメントは、終端ステートメントに到達するまで実行されます。 **DO** 構造体の範囲が実行されている間、増分値の処理以外では、**DO** 変数を再定義することも、未定義状態にすることもできません。

終端ステートメントの実行

終端ステートメントは、通常の実行順序の結果として、あるいは、制御の移動の結果として実行されます (この場合も **DO** 構造体の範囲外から範囲内へ、制御を移すことはできません)。終端ステートメントが実行された結果、制御が移動しない場合、増分値の処理に進みます。

増分値の処理

1. 最後に実行された **DO** ステートメントで始まるアクティブな **DO** 構造体の、**DO** 変数、繰り返し回数、および増分値が処理の対象として選択されます。
2. **DO** 変数の値が m_3 の値だけ増やされます。
3. 繰り返し回数から 1 が引かれます。
4. 繰り返し回数が 1 少なくなった **DO** 構造体の、ループ制御の処理に進みます。

マイグレーションのためのヒント:

GOTO ステートメントの代わりに、**EXIT** ステートメント、**CYCLE** ステートメント、および無限 **DO** ステートメントを使用します。

FORTRAN 77 ソース

```
        I = 0
        J = 0
20      CONTINUE
        I = I + 1
        J = J + 1
        PRINT *, I
        IF (I.GT.4) GOTO 10   ! Exiting loop
        IF (J.GT.3) GOTO 20   ! Iterate loop immediately
        I = I + 2
        GOTO 20
10      CONTINUE
        END
```

Fortran 90 または Fortran 95 ソース

```
        I = 0 ; J = 0
        DO
          I = I + 1
          J = J + 1
          PRINT *, I
          IF (I.GT.4) EXIT
          IF (J.GT.3) CYCLE
          I = I + 2
        END DO
        END
```

例:

```
INTEGER :: SUM=0
OUTER: DO
  INNER: DO
    READ (5,*) J
    IF (J.LE.I) THEN
      PRINT *, 'VALUE MUST BE GREATER THAN ', I
      CYCLE INNER
    END IF
    SUM=SUM+J
    IF (SUM.GT.500) EXIT OUTER
    IF (SUM.GT.100) EXIT INNER
  END DO INNER
  SUM=SUM+I
  I=I+10
END DO OUTER
PRINT *, 'SUM =',SUM
END
```

DO WHILE 構造体

DO WHILE 構造体は、**DO WHILE** ステートメントで指定されたスカラー論理式が真であり続ける限り、ステートメント・ブロックを繰り返し実行することを示します。**CYCLE** ステートメントを使用して特定の繰り返しを短縮したり、**EXIT** ステートメントを使用してループを終了することができます。

```
►►──DO_WHILE_statement──────────────────────────────────────────────►◄

►►──statement_block─────────────────────────────────────────────────►◄

►►──┬─END_DO_statement─────┬────────────────────────────────────────►◄
    └─terminal_statement───┘
```

DO_WHILE_statement
: 構文の詳細については、 337 ページの『DO WHILE』を参照してください。

END_DO_statement
: 構文の詳細については、 351 ページの『END (構造体)』を参照してください。

terminal_statement
: **DO WHILE** 構造体を終了するステートメントです。詳細については、 153 ページの『終端ステートメント』を参照してください。

DO 構造体の名前および範囲、アクティブ **DO** 構造体と非アクティブ **DO** 構造体、および終端ステートメントに関する前述の規則は、**DO WHILE** 構造体にも適用されます。

例

```
I=10
TWO_DIGIT: DO WHILE ((I.GE.10).AND.(I.LE.99))
  J=J+I
  READ (5,*) I
END DO TWO_DIGIT
END
```

分岐

分岐 によって通常の実行順序を変更することもできます。分岐は、同じ有効範囲単位内で、あるステートメントからラベル付きの分岐ターゲット・ステートメントに制御を移します。 **CASE**、**ELSE**、または **ELSE IF** 以外の実行可能ステートメントは、すべて分岐ターゲット・ステートメントに指定できます。

次のステートメントは、分岐で使用することができます。

- 割り当て **GO TO**

  これは、**ASSIGN** ステートメントでステートメント・ラベルを指定した実行可能ステートメントにプログラム制御を移します。構文の詳細については、 382 ページの『GO TO (割り当て)』を参照してください。

- 計算 **GO TO**

  これは、いくつかある実行可能ステートメントの内の 1 つに制御を移します。構文の詳細については、 384 ページの『GO TO (計算)』を参照してください。

- 無条件 **GO TO**

  これは、指定された実行可能ステートメントに制御を移します。構文の詳細については、 385 ページの『GO TO (無条件)』を参照してください。

- 算術 **IF**

  これは、算術式の計算に従って、3 つの実行可能ステートメントのうちの 1 つに制御を移します。 構文の詳細については、 386 ページの『IF (算術)』を参照してください。

次の I/O 指定子は、分岐で使用することができます。

- **END=** ファイル終わり指定子

  READ ステートメントで、ファイル終了レコードが検出された場合 (エラーが発生しない)、指定された実行可能ステートメントに制御を移します。

- **ERR=** エラー指定子

  エラーが発生した場合、指定された実行可能ステートメントに制御を移します。**BACKSPACE**、**ENDFILE**、**REWIND**、**CLOSE**、**OPEN**、**READ**、**WRITE**、および **INQUIRE** ステートメントで指定子を指定できます。

- **EOR=** レコード終わり指定子

  READ ステートメントで、レコードの終わり条件が検出された場合 (エラーが発生しない)、指定された実行可能ステートメントに制御を移します。

第 7 章 プログラム単位およびプロシージャー

この章では、以下の項目について説明します。

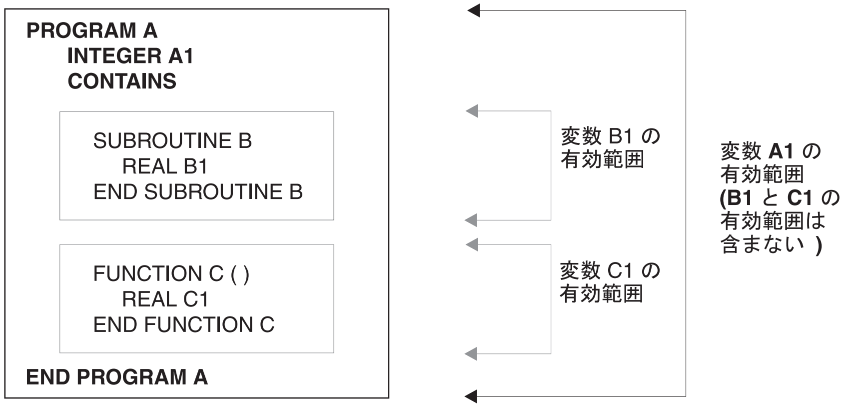
- 『有効範囲』
- 164 ページの『関連付け』
- 169 ページの『プログラム単位、プロシージャー、およびサブプログラム』
- 172 ページの『インターフェース・ブロック』
- 176 ページの『総称インターフェース・ブロック』
- 181 ページの『メインプログラム』
- 183 ページの『モジュール』
- 187 ページの『ブロック・データのプログラム単位』
- 188 ページの『関数およびサブルーチン・サブプログラム』
- 191 ページの『組み込みプロシージャー』
- 193 ページの『引き数』
- 196 ページの『引き数関連付け』
- 209 ページの『再帰』
- F95 209 ページの『純粋プロシージャー』 F95
- F95 212 ページの『エレメント型プロシージャー』 F95

有効範囲

プログラム単位は、一連のオーバーラップしない有効範囲単位から構成されています。*有効範囲単位* とは、固有の有効範囲の境界を持つプログラム単位の部分です。それは次のうちの 1 つです。

- 派生型定義
- プロシージャー・インターフェース本体 (その中の定義およびインターフェース本体は含まない)
- プログラム単位、モジュール・サブプログラム、または内部サブプログラム (その中の定義およびインターフェース本体、モジュール・サブプログラム、内部サブプログラムは含まない)

ホスト有効範囲単位 とは、別の有効範囲単位を直接包含している有効範囲単位のことです。たとえば、次の図では、内部関数 C のホスト有効範囲単位は、メインプログラム A の有効範囲単位です。ホスト関連付けは、内部サブプログラム、モジュール・サブプログラム、または定義がそのホストから名前にアクセスできるようにします。



エンティティーの有効範囲は次のとおりです。

- 名前 (下記を参照)
- ラベル (ローカル・エンティティー)
- 外部 I/O 装置番号 (グローバル・エンティティー)
- 演算子記号。組み込み演算子はグローバル・エンティティーで、定義済み演算子はローカル・エンティティーです。
- 割り当て記号 (グローバル・エンティティー)

有効範囲が実行可能プログラムである場合、そのエンティティーは*グローバル・エンティティー* と呼ばれます。有効範囲が有効範囲単位である場合、そのエンティティーは*ローカル・エンティティー* と呼ばれます。有効範囲がステートメント、またはステートメントの一部の場合、そのエンティティーは*ステートメント・エンティティー* と呼ばれます。 F95 有効範囲が構造体である場合、そのエンティティーは、*構造体エンティティー* と呼ばれます。 F95

名前の有効範囲

グローバル・エンティティー

IBM 拡張

グローバル・エンティティーには、プログラム単位、外部プロシージャー、共通ブロック、および **CRITICAL** *lock_names* があります。

IBM 拡張 の終り

ある名前で 1 つのグローバル・エンティティーを識別する場合には、その名前を使用して同じ実行可能プログラムにある別のグローバル・エンティティーを識別することはできません。

グローバル・エンティティーの名前に関する制約事項の詳細については、「ユーザーズ・ガイド」の『*XL Fortran 外部名の規則*』を参照してください。

ローカル・エンティティー

以下のクラスのエンティティーは、そのエンティティーが定義されている有効範囲単位のローカル・エンティティーになります。

1. 名前付き変数のうち、ステートメント・エンティティー、モジュール・プロシージャー、名前付き定数、定義、構造体名、総称識別子、ステートメント関数、内部サブプログラム、仮プロシージャー、組み込みプロシージャー、または名前リスト・グループ名以外のもの。
2. 派生型定義のコンポーネント (派生型定義にはそれぞれ固有のクラスがあります)。

   コンポーネント名は、コンポーネントのタイプと同じ有効範囲を持ちます。コンポーネントは、そのタイプの構造体のコンポーネント指定子内にのみ指定されます。

   派生型がモジュール内で定義され、 **PRIVATE** ステートメントが指定されていると、タイプおよびそのコンポーネントは、ホスト関連付けによって定義モジュールのサブプログラム内でアクセス可能となります。アクセス元の有効範囲単位が、使用関連付けによってこのタイプにアクセスした場合、その有効範囲単位 (およびホスト関連付けによってその有効範囲単位のエンティティーにアクセスするすべての有効範囲単位) は、派生型定義にはアクセスできますが、そのコンポーネントにはアクセスできません。
3. 引き数キーワード (明示インターフェースを使用した各プロシージャー用の個別のクラス)

   内部プロシージャー、モジュール・プロシージャー、またはプロシージャーのインターフェース・ブロックにある仮引き数名は、そのホストの有効範囲単位の引き数キーワードが有効範囲となっています。引き数キーワードとしての仮引き数名は、仮引き数であるプロシージャーに対するプロシージャー参照でのみ指定できます。使用関連付けまたはホスト関連付けによって、プロシージャーまたはプロシージャーのインターフェース・ブロックが別の有効範囲単位内でアクセス可能な場合、引き数キーワードはその有効範囲単位内のプロシージャーに対するプロシージャー参照についてアクセス可能です。

ある有効範囲単位内では、1 つのクラスのローカル・エンティティーを識別する名前を使用して、別のクラスのローカル・エンティティーを識別することができます。 この名前は、総称名の場合を除いて、同じクラスのローカル・エンティティーを識別するために使用することはできません。また、ある有効範囲単位内のグローバル・エンティティーを識別する名前を使用して、その有効範囲単位内のクラス 1 のローカル・エンティティーを識別することもできません。ただし、共通ブロック名または外部関数名の場合は除きます。レコード構造のコンポーネントは、クラス 2 のローカル・エンティティーです。タイプごとに別個のクラスが存在します。

レコード構造宣言を使用して派生型になるよう宣言された名前には、派生型以外のその有効範囲単位のクラス 1 の別のローカル・エンティティーと同じ名前を付けることがで

きます。この場合は、そのタイプの構造体コンストラクターは、その有効範囲では利用不能です。同様に、クラス 1 のローカル・エンティティーには、その有効範囲内にアクセス可能なクラス 1 の別のローカル・エンティティーがある場合でも、ホスト関連付けまたは使用関連付けを介してアクセス可能です。

- 2 つのエンティティーのうちの 1 つが派生型で、もう一方がそうでない場合で、
- ホスト関連付けの場合、派生型はホスト関連付けを介してアクセス可能です。たとえば、モジュール M、プログラム単位 P、および P にネストされた内部サブプログラムまたはモジュール・サブプログラム S があるとします。 P 内 (または S 内) の関連付けを使用してアクセスされる M で宣言された T1 という名前のエンティティーがある場合は、 2 つのうち 1 つが派生型である場合に限って、 T1 と同じ名前で別のエンティティーを P 内に (または S 内にそれぞれ) 宣言できます。 P 内にアクセス可能な T2 という名前のエンティティーがあり、 S で宣言された T2 という名前のエンティティーがある場合、 P 内でアクセス可能な T2 は、P 内の T2 が派生型の場合は、S 内でもアクセス可能です。P 内の T2 が派生型でない場合は、 S が別の T2 (派生型または派生型でないもの) を宣言した場合でも、S 内でアクセスできません。

このタイプの構造体コンストラクターは、その有効範囲では利用不能です。その有効範囲でアクセス可能な派生型と同じ名前の付いている有効範囲のクラス 1 のローカル・エンティティーは、その有効範囲の宣言ステートメントに明示的に宣言される必要があります。

クラス 1 の 2 つのローカル・エンティティーのうち 1 つが派生型であり有効範囲単位内にアクセス可能な場合は、エンティティーの名前を指定する **PUBLIC** または **PRIVATE** ステートメントのいずれかは、両方のエンティティーに適用されます。エンティティーの名前が **VOLATILE** ステートメントに指定される場合、そのエンティティー、またはその有効範囲に宣言されたエンティティーは、揮発属性を持ちます。2 つのエンティティーがモジュールの共通エンティティーの場合、そのモジュールを参照し、*use_name* としてエンティティーの名前を指定する **USE** ステートメント上での名前変更は、両方のエンティティーに適用されます。

有効範囲単位内の共通ブロック名は、名前付き定数または組み込みプロシージャー以外であれば、どのローカル・エンティティーの名前であってもかまいません。この名前は、**COMMON**、**VOLATILE**、**SAVE** ステートメントでスラッシュで区切られる場合にのみ、共通ブロック・エンティティーとして認識されます。これ以外の場合、その名前はローカル・エンティティーを識別します。組み込みプロシージャー名は、組み込みプロシージャーを参照しない有効範囲単位内の共通ブロック名である可能性があります。この場合、組み込みプロシージャー名にはアクセスできません。

外部関数名を関数結果名とすることもできます。これは、外部関数名をローカル・エンティティーとする唯一の方法です。

有効範囲単位に組み込みプロシージャーと同じ名前のクラス 1 のローカル・エンティティーが含まれている場合、その有効範囲単位にある組み込みプロシージャーにはアクセスできません。

インターフェース・ブロックの総称名は、そのインターフェース・ブロックにあるプロシージャー名のいずれか、つまりアクセス可能な総称名と同じ名前の場合があります。また、総称組み込みプロシージャーと同じ名前である可能性もあります。詳細については、 206 ページの『プロシージャー参照の解決』を参照してください。

ステートメント・エンティティーおよび構造体エンティティー

ステートメント・エンティティー: 以下の項目は、ステートメント・エンティティーです。

- ステートメント関数の仮引き数の名前
  有効範囲: 名前を指定するステートメントの有効範囲
- **DATA** ステートメントまたは配列コンストラクターにある暗黙 **DO** の **DO** 変数として指定する変数の名前
  有効範囲: 暗黙 **DO** リストの有効範囲

共通ブロック名またはスカラー変数名を除き、ステートメントまたは構造体の有効範囲単位内でアクセス可能なクラス 1 のグローバル・エンティティーまたはローカル・エンティティーの名前は、そのステートメントまたは構造体のそれぞれのエンティティー名と同じにすることはできません。ステートメント・エンティティーまたは構造体エンティティーの有効範囲内で、別のステートメント・エンティティーまたは構造体エンティティーが同じ名前を持つことはできません。

ステートメント関数のステートメントに仮引き数として指定される変数名の有効範囲は、指定されているステートメントの有効範囲と同じです。この変数名のタイプおよび型付きパラメーターは、有効範囲単位内のステートメント関数を含む変数名である場合に持つタイプおよび型付きパラメーターと同じです。

ステートメントまたは構造体の有効範囲単位内でアクセス可能なグローバル・エンティティーまたはローカル・エンティティーの名前が、そのステートメントまたは構造体のステートメント・エンティティーまたは構造体エンティティーと同じ名前の場合、その名前はステートメント・エンティティーまたは構造体エンティティーの有効範囲として、そのステートメント・エンティティーまたは構造体エンティティーの有効範囲内で解釈されます。ステートメント・エンティティーまたは構造体エンティティーの有効範囲外にあるステートメントまたは構造体の一部を含む有効範囲単位内の別の場所では、名前はグローバル・エンティティーまたはローカル・エンティティーの有効範囲として解釈されます。

ステートメント・エンティティーまたは構造体エンティティーの名前が変数、名前付き定数、または関数を示すアクセス可能な名前と同じ場合、そのステートメント・エンティティーまたは構造体エンティティーのタイプおよび型付きパラメーターはその変数、

名前付き定数、または関数と同じになります。名前が同じでない場合のステートメント・エンティティーまたは構造体エンティティーのタイプは、有効な暗黙のタイプ設定規則によって決まります。ステートメント・エンティティーが **DATA** ステートメントにある暗黙 **DO** の **DO** 変数である場合、その変数にはアクセス可能な名前付き定数と同じ名前を付けることはできません。

Fortran 95 ステートメントおよび構造体エンティティー:

Fortran 95

以下は、Fortran 95 ステートメントおよび構造体エンティティーです。

- **FORALL** ステートメントまたは **FORALL** 構造体の *index_name* として指定する変数の名前
  有効範囲: **FORALL** ステートメントまたは構造体の有効範囲

FORALL ステートメントまたは構造体エンティティーが持つ属性は、 **FORALL** を含む有効範囲単位内の変数名がそのエンティティーの名前である場合に持つことになるタイプおよび型付きパラメーターだけであり、整数タイプになります。

共通ブロック名またはスカラー変数名を除き、クラス 1 のグローバル・エンティティーおよびローカル・エンティティーを識別する名前 (**FORALL** ステートメントまたは **FORALL** 構造体の有効範囲単位内でアクセス可能) は、*index_name* と同じにすることはできません。 **FORALL** 構造体の有効範囲単位内で、ネストされた **FORALL** ステートメントまたは **FORALL** 構造体には、同じ *index_name* を付けることはできません。

FORALL ステートメントまたは **FORALL** 構造体の有効範囲単位内でアクセス可能なグローバル・エンティティーまたはローカル・エンティティーの名前が *index_name* と同じ場合、その名前は **FORALL** ステートメントまたは **FORALL** 構造体の有効範囲内では *index_name* として解釈されます。 また有効範囲単位内のどこにおいても、その名前はグローバル・エンティティーまたはローカル・エンティティーの名前として解釈されます。

Fortran 95 の終り

関連付け

関連付けは、同一の有効範囲単位内で同一のデータ項目を複数の異なる名前で識別する場合、または同一の実行可能プログラムの異なる有効範囲単位で、同一のデータ項目を同じ名前または複数の異なる名前で識別する場合に生じます。

ホスト関連付け

ホスト関連付け を使用すれば、内部サブプログラム、モジュール・サブプログラム、または派生型定義からそのホスト内の名前付きエンティティーにアクセスできるようにな

ります。アクセスされるエンティティーは、ホスト内のときと同じ属性を持ち、 (もしあれば) 同じ名前で識別されます。上記エンティティーには、オブジェクト、派生型定義、名前リスト・グループ、インターフェース・ブロックおよびプロシージャーなどが含まれます。

EXTERNAL 属性が指定されている名前はグローバル名です。この名前を非総称名として持つホスト有効範囲単位内のどのエンティティーにも、その名前およびホスト関連付けを使用してアクセスすることはできません。

ある有効範囲単位内で宣言または初期化が行われる場合、次に示すエンティティーはその有効範囲単位内でローカルとなります。
- **COMMON** ステートメント内の変数名または **DATA** ステートメント内で初期化された変数名
- **DIMENSION** ステートメント内の配列名
- 派生型の名前
- タイプ宣言内のオブジェクト名、**EQUIVALENCE**、**POINTER**、**ALLOCATABLE**、**SAVE**、**TARGET**、**AUTOMATIC**、整数 **POINTER**、**STATIC**、または **VOLATILE** ステートメント
- **PARAMETER** ステートメント内の名前付き定数
- **NAMELIST** ステートメント内の名前リスト・グループ名
- インターフェース総称名または定義済み演算子
- **INTRINSIC** ステートメント内の組み込みプロシージャー名
- **FUNCTION** ステートメント、ステートメント関数ステートメント、またはタイプ宣言ステートメント内の関数名
- **FUNCTION** ステートメントまたは **ENTRY** ステートメント内の結果名
- **SUBROUTINE** ステートメント内のサブルーチン名
- **ENTRY** ステートメント内のエントリー名
- **FUNCTION** ステートメント、 **SUBROUTINE** ステートメント、**ENTRY** ステートメント、またはステートメント関数ステートメント内の仮引き数名
- 名前付き構造体の名前

サブプログラムに対してローカルであるエンティティーは、ホスト有効範囲単位内ではアクセスできません。

次のような場合、**DATA** ステートメントの前にローカル・エンティティーの参照または定義を行うことはできません。
1. ある有効範囲単位に限定してローカルである場合。エンティティーは **DATA** ステートメントで初期化されるからです。
2. ホストにあるエンティティーがローカル・エンティティーの名前と同じ場合。

ホストの派生型名にアクセスできない場合でも、そのタイプの構造体またはそのような構造体のサブオブジェクトにはアクセス可能です。

サブプログラムがホスト関連付けを介してポインター (または複数ポインター) にアクセスすると、サブプログラムの呼び出し時に存在するポインター関連付けはサブプログラムに現行のまま残ります。このポインター関連付けは、サブプログラム内で変更してもかまいません。プロシージャーが実行を終了しても、ポインター関連付けは現行のまま残ります。ただし、これによってポインターが未定義になる場合は例外です。この場合、ホストに関連したポインターの関連付け状況は未定義となります。

インターフェース本体は、使用関連付けを介してエンティティーを使用することはできますが、ホスト関連付けを介して名前付きエンティティーを使用することはありません。

内部サブプログラムまたはモジュール・サブプログラムのホスト有効範囲単位には、同一の使用関連付けをもったエンティティーを指定することができます。

ホスト関連付けの例

```
SUBROUTINE MYSUB
TYPE DATES                   ! Define DATES
  INTEGER START
  INTEGER END
END TYPE DATES
CONTAINS
  INTEGER FUNCTION MYFUNC(PNAME)
  TYPE PLANTS
    TYPE (DATES) LIFESPAN    ! Host association of DATES
      CHARACTER(10) SPECIES
      INTEGER PHOTOPER
    END TYPE PLANTS
  END FUNCTION MYFUNC
END SUBROUTINE MYSUB
```

使用関連付け

使用関連付け は、**USE** ステートメントによってモジュールのエンティティーにアクセスする際に生じます。使用に関連したエンティティーの名前を変更して、ローカルの有効範囲単位内で使用することが可能です。この関連付けは実行可能プログラムの実行時の間は有効です。詳細については、488 ページの『USE』を参照してください。

```
MODULE M
  CONTAINS
  SUBROUTINE PRINTCHAR(X)
    CHARACTER(20) X
    PRINT *, X
  END SUBROUTINE
END MODULE
PROGRAM MAIN
USE M                         ! Accesses public entities of module M
CHARACTER(20) :: NAME='George'
CALL PRINTCHAR(NAME)          ! Calls PRINTCHAR from module M
END
```

ポインター関連付け

ポインターに関連したターゲットは、ポインターに対する参照によって、参照することができます。これを*ポインター関連付け*と呼びます。

ポインターはすべて次のような関連付け状況となります。

関連 (Associated)

- **ALLOCATE** ステートメントによって正常にポインターが割り振られ、後で関連解除 (Disassociated) されたり、未定義 (Undefined) にされることはありませんでした。

```
ALLOCATE (P(3))
```

- これは現在関連しているか、または **TARGET** 属性を有しているターゲットに割り当てられているポインターです。割り当て可能である場合には、現在割り振られているポインターを示しています。

```
P => T
```

関連解除 (Disassociated)

- ポインターは、**NULLIFY** ステートメントまたは **-qinit=f90ptr** オプションによってヌル文字化されています。「ユーザーズ・ガイド」の 『**-qinit**』を参照してください。

```
NULLIFY (P)
```

- ポインターの割り振りが正常に解除されています。

```
DEALLOCATE (P)
```

- これは関連解除されたポインターに割り当てられているポインターです。

```
NULLIFY
(Q); P => Q
```

未定義 (Undefined)

- 初期の状態 (**-qinit=f90ptr** オプションが指定されていない場合)
- ターゲットが割り振られていない場合
- ポインター以外によってターゲットの割り振りが解除された場合

```
POINTER
P(:), Q(:)
ALLOCATE (P(3))
Q => P
DEALLOCATE (Q)   ! Deallocate target of P through Q.
                 ! P is now undefined.
END
```

- **RETURN** ステートメントまたは **END** ステートメントの実行により、ポインターのターゲットが未定義になった場合
- ポインターが宣言またはアクセスされたプロシージャー内での **RETURN** ステートメントまたは **END** ステートメントの実行後。ただし、74 ページの『未定義を発生させるイベント』の項目 4 に記述したオブジェクトを除きます。

定義状況および関連付け状況

ポインターの定義状況は、そのターゲットの状況を示します。ポインターが定義可能なターゲットに関連している場合、そのポインターの定義状況は、変数の規則にしたがって定義または未定義状態にすることができます。

ポインターの関連付け状況が関連解除 (Disassociated) または未定義 (Undefined) の場合には、そのポインターを参照したり割り振りを解除できません。関連付け状況がどんな場合でも、ポインターは必ずヌル文字化され、割り振られるかまたはポインター割り当てをされます。ポインターが割り振られると、その定義状況は未定義になります。ポインターがポインター割り当てをされると、その関連付け状況および定義状況は、そのターゲットによって決められます。したがって、ポインターが定義されたターゲットに関連すると、そのポインターは定義された状態になります。

整数ポインター関連付け

IBM 拡張

データ・オブジェクトに関連した整数ポインターを使用すると、データ・オブジェクトを参照することが可能になります。 これを*整数ポインター関連付け*と呼びます。

整数ポインター関連付けは、次のような状況でのみ生じます。

- 整数ポインターに、変数のアドレスを割り当てた場合

```
      POINTER
(P,A)
      P=LOC(B)                ! A and B become associated
```

- 複数のポインティング先を同一の整数ポインターを指定して宣言した場合

```
POINTER (P,A), (P,B)   ! A and B are associated
```

- 複数の整数ポインターに、同じ変数のアドレスまたはストレージに関連した他の変数のアドレスを割り当てた場合

```
POINTER (P,A), (Q,B)
      P=LOC(C)
      Q=LOC(C)                ! A, B, and C become associated
```

- 仮引き数として指定した整数ポインター変数に、他の仮引き数または共通ブロックのメンバーのアドレスを割り当てた場合

```
      POINTER (P,A)
          .
          .
      CALL SUB (P,B)
          .
          .
      SUBROUTINE SUB (P,X)
```

```
POINTER (P,Y)
P=LOC(X)                ! Main program variables A
                        !   and B become associated.
```

IBM 拡張 の終り

プログラム単位、プロシージャー、およびサブプログラム

プログラム単位は 1 行以上の文字列で、ステートメント、注釈、および **INCLUDE** ディレクティブで構成されています。特に以下のものを指します。

- メインプログラム
- モジュール
- ブロック・データのプログラム単位
- 外部関数サブプログラム
- 外部サブルーチン・サブプログラム

実行可能プログラムは、1 つのメインプログラム、任意の数の外部サブプログラム、モジュール、およびブロック・データのプログラム単位からなるプログラム単位の集合体です。

メインプログラムまたはサブプログラムから、あるサブプログラムを呼び出して特定の処理を行わせることができます。プロシージャーを呼び出すと、参照されたサブプログラムが実行されます。

外部サブプログラムまたはモジュール・サブプログラムには、複数の **ENTRY** ステートメントが入っています。サブプログラムは、 **ENTRY** ステートメントごとに 1 つのプロシージャーを定義するのはもとより **SUBROUTINE** または **FUNCTION** ステートメント用に 1 つのプロシージャーを定義します。

外部プロシージャーは、外部サブプログラムまたは Fortran 以外のプログラム言語で作成されたプログラム単位のいずれかで定義されます。

メインプログラム、外部プロシージャー、ブロック・データのプログラム単位、およびモジュールの名前は、グローバル・エンティティーです。内部プロシージャーおよびモジュール・プロシージャーの名前は、ローカル・エンティティーです。

内部プロシージャー

外部サブプログラム、モジュール・サブプログラム、およびメインプログラムには、関数としてであれサブルーチンとしてであれ、 **CONTAINS** ステートメント以降に内部サブプログラムを組み込むことができます。

内部プロシージャーは、内部サブプログラムで定義します。内部サブプログラムは他の内部サブプログラムには組み込めません。 モジュール・プロシージャーは、モジュール・サブプログラムまたはモジュール・サブプログラム内のエントリーで定義します。

内部プロシージャーおよびモジュール・プロシージャーは、次の点を除いて外部プロシージャーと同じです。

- 内部プロシージャーまたはモジュール・プロシージャーの名前は、グローバル・エンティティーではありません。
- 内部サブプログラムは、**ENTRY** ステートメントを含むことはできません。
- 内部プロシージャー名を仮プロシージャーと関連する引き数にすることはできません。
- 内部サブプログラムまたはモジュール・サブプログラムは、ホスト関連付けを介してホスト・エンティティーへのアクセス権を持っています。

マイグレーションのためのヒント:

外部プロシージャーを内部サブプログラムに変えるか、またはモジュールの中に入れます。明示的インターフェースは、タイプの検査を行います。

FORTRAN 77 ソース

```
PROGRAM MAIN
  INTEGER A
  A=58
  CALL SUB(A)
C A MUST BE PASSED
END
SUBROUTINE SUB(A)
  INTEGER A,B,C   ! A must be redeclared
  C=A+B
END SUBROUTINE
```

Fortran 90 または Fortran 95 ソース

```
PROGRAM MAIN
  INTEGER :: A=58
  CALL SUB
  CONTAINS
  SUBROUTINE SUB
    INTEGER B,C
    C=A+B         ! A is accessible by host association
  END SUBROUTINE
END
```

インターフェースの概念

プロシージャーのインターフェースは、プロシージャー参照の形式を決定します。インターフェースは次の部分からなります。

- プロシージャーの特性
- プロシージャーの名前
- 仮引き数の名前および特性
- プロシージャーの総称識別子 (ある場合)

プロシージャーの特性には次のようなものがあります。

- サブルーチンまたは関数としてプロシージャーを区別すること
- 各仮引き数をデータ・オブジェクト、ダミー・プロシージャー、または選択戻り指定子としていずれかを区別すること

  仮のデータ・オブジェクトの特性には、そのタイプ、型付きパラメーター (ある場合)、形状、意図、オプションであるかどうか、割り振り可能かどうか、ポインターであるかどうか、およびターゲットであるかどうかなどがあります。タイプ・パラメーターまたは配列境界の決定に関して他のオブジェクトにどの程度左右されるかも特性の 1 つです。形状、サイズ、および文字長が想定される場合、それも特性に含まれます。

  仮プロシージャーの特性には、インターフェースの明示性、プロシージャーの特性 (インターフェースが明示的な場合)、およびオプションかどうかなどがあります。
- プロシージャーが関数の場合、タイプ、型付きパラメーター (ある場合)、ランク、割り振り可能かどうか、およびポインターであるかどうかなど、結果値の特性を指定します。非ポインターの配列結果の場合、その形状も特性の 1 つです。型付きパラメーターまたは配列境界の決定に関して他のオブジェクトにどの程度左右されるかも特性の 1 つです。文字オブジェクトの長さが想定される場合は、それも特性に含まれます。

有効範囲単位内でプロシージャーにアクセス可能であれば、プロシージャーはその有効範囲単位内で明示的または暗黙のどちらかのインターフェースをとります。この場合の規則は次のとおりです。

エンティティー	**インターフェース**
仮プロシージャー	インターフェース・ブロックがあるか、または インターフェース・ブロックにアクセス可能な場合、有効範囲単位内で明示的 その他のすべての場合は暗黙
外部サブプログラム	インターフェース・ブロックがあるか、または インターフェース・ブロックにアクセス可能な場合、有効範囲単位内で明示的 その他のすべての場合は暗黙
結果文節付きの再帰的プロシージャー	サブプログラムの当該有効範囲単位内で明示的
モジュール・プロシージャー	常に明示的
内部プロシージャー	常に明示的
総称プロシージャー	常に明示的
組み込みプロシージャー	常に明示的
ステートメント関数	常に暗黙

内部サブプログラムはインターフェース・ブロックには入れられません。

範囲単位指定内のプロシージャーは、アクセス可能な複数のインターフェースを持つことはできません。

ステートメント関数のインターフェースをインターフェース・ブロック内に指定することはできません。

明示的インターフェース

次の場合、プロシージャーは明示的インターフェースが必要となります。

1. 次の条件でプロシージャーへの参照がある場合
   - 引き数キーワードを指定したプロシージャーへの参照
   - 定義済み割り当てとしてのプロシージャーへの参照 (サブルーチンの場合のみ)
   - 定義済み演算子としての式内でのプロシージャーへの参照 (関数の場合のみ)
   - 総称名による参照としてのプロシージャーへの参照
   - F95 純粋であることが要求されるコンテキスト内 F95
2. プロシージャーに次のものがある場合
   - オプションの仮引き数
   - 配列値の結果 (関数の場合のみ)
   - ポインター、ターゲット、割り振り可能、または想定形状配列である仮引き数
   - length 型付きパラメーターが想定または定数のいずれでもない結果 (文字関数の場合のみ)
   - ポインターまたは割り振り可能の結果 (関数の場合のみ)
3. F95 プロシージャーはエレメント型プロシージャーです。 F95

暗黙インターフェース

インターフェースが明確でない場合、つまりサブプログラムに明示インターフェースがない場合、プロシージャーは暗黙のインターフェースをとります。

インターフェース・ブロック

インターフェース・ブロック は、外部プロシージャーおよび仮プロシージャー用の明示的インターフェースを指定する 1 つの手段です。また、このインターフェース・ブロックを使用して、総称識別子を定義することもできます。インターフェース・ブロック内のインターフェース本体 とは、既存の外部プロシージャーまたは仮プロシージャー用の特定の明示的インターフェースを指定するものです。

```
>>──INTERFACE_statement──────────────────────────────────────────><

     ┌──────────────────────────────┐
     ▼                              │
>>────┬─FUNCTION_interface_body────┬┴──────────────────────────────><
      ├─SUBROUTINE_interface_body──┤
      └─MODULE_PROCEDURE_statement─┘

>>──END_INTERFACE_statement──────────────────────────────────────><
```

INTERFACE_statement
: 構文の詳細については、 407 ページの『INTERFACE』を参照してください。

END_INTERFACE_statement
: 構文の詳細については、 354 ページの『END INTERFACE』を参照してください。

MODULE_PROCEDURE_statement
: 構文の詳細については、 417 ページの『MODULE PROCEDURE』を参照してください。

FUNCTION_interface_body

```
>>──FUNCTION_statement───────────────────────────────────────────><

>>──┬────────────────────┬───────────────────────────────────────><
    └─specification_part─┘

>>──end_function_statement───────────────────────────────────────><
```

SUBROUTINE_interface_body

```
▶▶─SUBROUTINE_statement──────────────────────────────────────────────▶◀

▶▶──┬────────────────────┬────────────────────────────────────────────▶◀
    └─specification_part─┘

▶▶─end_subroutine_statement──────────────────────────────────────────▶◀
```

FUNCTION_statement、*SUBROUTINE_statement*
: 構文の詳細については、 378 ページの『FUNCTION』および 473 ページの『SUBROUTINE』を参照してください。

specification_part
: 24 ページの『ステートメントおよび実行の順序』で、 **2** および **4** の番号が付けられたステートメント・グループの順序列です。

end_function_statement、*end_subroutine_statement*
: 構文の詳細については、 350 ページの『END』を参照してください。

インターフェース本体においては、プロシージャーの特性すべてを指定してください。 170 ページの『インターフェースの概念』を参照してください。その特性は、サブプログラムの定義で指定した内容と一致している必要があります。ただし、以下の場合を除きます。

1. 仮引き数の名前は一致しなくてもかまいません。
2. プロシージャーは、純粋であると定義しているサブプログラムであるとしても、純粋であることを示す必要はありません。
3. 純粋な実引き数を、純粋でない仮プロシージャーと関連付けることができます。
4. 組み込みエレメント型プロシージャーを仮プロシージャーと関連付ける場合、仮プロシージャーはエレメント型プロシージャーである必要はありません。

インターフェース本体の *specification_part* は、プロシージャーの特性を決定しないデータ・オブジェクトの属性指定や値の定義を行うステートメントを含めることが可能です。そのような仕様ステートメントは、インターフェースに影響を及ぼしません。インターフェース・ブロックは、その特性がモジュール・サブプログラム定義で定義されるモジュール・プロシージャーの特性を指定しません。

インターフェース本体には、**ENTRY** ステートメント、 **DATA** ステートメント、**FORMAT** ステートメント、ステートメント関数ステートメント、または実行可能ステートメントを入れることはできません。エントリー・インターフェースを指定することにより、エントリー名をインターフェース本体内のプロシージャー名として使用することができます。

インターフェース本体は、ホスト関連付けを介して名前付きエンティティーを使用することはありません。インターフェース本体はデフォルトの暗黙規則付きのホストと同様に扱われます。暗黙規則の説明については、 69 ページの『タイプの決め方』を参照してください。

インターフェース・ブロックは、総称または非総称のいずれかです。総称インターフェース・ブロックは、**INTERFACE** ステートメント内で総称仕様を指定しなければなりませんが、非総称インターフェース・ブロックではそのような総称仕様を指定する必要はありません。詳細については、 407 ページの『INTERFACE』を参照してください。

非総称インターフェース・ブロック内部のインターフェース本体には、サブルーチンおよび関数双方のインターフェースを入れることができます。

総称名は、インターフェース・ブロック内のすべてのプロシージャーを参照するために、単一の名前を指定します。総称名でのプロシージャー参照が発生するたびに、最大 1 つの特定のプロシージャーが呼び出されます。

MODULE PROCEDURE ステートメントが使用できるのは、インターフェース・ブロックが総称仕様が指定されていて、かつ各プロシージャー名がモジュール・プロシージャーとしてアクセス可能な有効範囲単位内に入れられている場合だけです。

MODULE PROCEDURE ステートメントで使用されるプロシージャー名は、アクセス可能なインターフェース・ブロック中の **MODULE PROCEDURE** ステートメントでこれまでに同一の総称識別子を指定して指定された名前にすることはできません。

IBM 拡張

Fortran 以外のサブプログラムに対するインターフェースの場合、**FUNCTION** または **SUBROUTINE** ステートメント内の仮引き数リストにより引き渡し方法を明示的に指定できます。詳細については、 195 ページの『仮引き数』を参照してください。

IBM 拡張 の終り

インターフェースの例

```
MODULE M
 CONTAINS
 SUBROUTINE S1(IARG)
   IARG = 1
 END SUBROUTINE S1
 SUBROUTINE S2(RARG)
   RARG = 1.1
 END SUBROUTINE S2
 SUBROUTINE S3(LARG)
```

```
   LOGICAL LARG
   LARG = .TRUE.
 END SUBROUTINE S3
END
```

```
USE M
INTERFACE SS
  SUBROUTINE SS1(IARG,JARG)
  END SUBROUTINE
  MODULE PROCEDURE S1,S2,S3
END INTERFACE
CALL SS(II)              ! Calls subroutine S1 from M
CALL SS(I,J)             ! Calls subroutine SS1
END
```

```
SUBROUTINE SS1(IARG,JARG)
  IARG = 2
  JARG = 3
END SUBROUTINE
```

特定のインターフェースを介してプロシージャーを必ず参照できます。プロシージャーに総称インターフェースが存在する場合は、総称インターフェースを介してプロシージャーを参照することができます。

インターフェース本体内では、仮引き数を仮プロシージャーとする場合、仮引き数を **EXTERNAL** 属性に指定するか、または仮引き数のインターフェースを確保する必要があります。

総称インターフェース・ブロック

総称インターフェース・ブロックでは、 **INTERFACE** ステートメントに総称名、定義済み演算子、または定義済み割り当てを指定する必要があります。 総称名は単一名で、その名前を使用することによりインターフェース・ブロックで指定されているすべてのプロシージャーを参照することができます。 総称名は、アクセス可能な総称名、またはインターフェース・ブロック内のプロシージャー名のどれかと同じ可能性があります。

ある有効範囲単位内でアクセス可能な複数の総称インターフェースが同じローカル名を持っている場合、それらの総称インターフェースは単一の総称インターフェースと見なされます。

明白な総称プロシージャー参照

総称プロシージャー参照がなされるとき、特定のプロシージャーが 1 つだけ呼び出されます。次の規則に従えば、総称参照は明白となります。

同一の有効範囲単位内にある 2 つのプロシージャーが共に、割り当てを定義しているか、定義済みの演算子および同数の引き数を持っている場合、異なるタイプ、kind 型付きパラメーター、またはランクを持つ仮引き数に引き数リスト内の位置で対応するもう 1 つの仮引き数を指定しなければなりません。

1 つの有効範囲単位内では、同じ総称名を持つ 2 つのプロシージャーは、両方ともサブルーチンであるか、または両方とも関数でなければなりません。また、2 つのうちの少なくとも 1 つに、次の両方の条件を満たす仮引き数が必須となります。

1. 他のサブプログラムの引き数リストにない仮引き数、あるいは異なるタイプ、異なる kind 型付きパラメーター、または異なるランクでリストされている仮引き数と、引き数リスト内の位置で対応するもの。
2. 他の引き数リストにない仮引き数、あるいは異なるタイプ、異なる kind 型付きパラメーター、または異なるランクでリストされている仮引き数と、引き数キーワードで対応するもの。

インターフェース・ブロックが組み込みプロシージャーを展開する場合 (次の項を参照)、その組み込みプロシージャーは、使用できる引き数の集合ごとに単一のプロシージャーが相当する特定のプロシージャーの集合から構成されている場合と同様に、前述の規則が適用されます。

IBM 拡張

注:

1. **BYTE** タイプの仮引き数と、**INTEGER(1)**、**LOGICAL(1)**、および文字タイプの対応する 1 バイトの仮引き数は同じであると見なされます。
2. **-qintlog** コンパイラー・オプションを指定すると、整数タイプおよび論理タイプの仮引き数と、同じ kind 型付きパラメーターを持つ整数タイプおよび論理タイプの対応する仮引き数のタイプは同じであると見なされます。
3. インターフェース本体内の **EXTERNAL** 属性を指定して仮引き数の宣言のみを行う場合、その仮引き数は位置と引き数キーワードでプロシージャーに対応する唯一の仮引き数となります。

IBM 拡張 の終り

総称インターフェース・ブロックの例

```
PROGRAM MAIN
INTERFACE A
  FUNCTION AI(X)
    INTEGER AI, X
  END FUNCTION AI
END INTERFACE
INTERFACE A
  FUNCTION AR(X)
    REAL AR, X
  END FUNCTION AR
END INTERFACE
INTERFACE FUNC
  FUNCTION FUNC1(I, EXT)      ! Here, EXT is a procedure
    INTEGER I
    EXTERNAL EXT
```

```
  END FUNCTION FUNC1
  FUNCTION FUNC2(EXT, I)
    INTEGER I
    REAL EXT                  ! Here, EXT is a variable
  END FUNCTION FUNC2
END INTERFACE
EXTERNAL MYFUNC
IRESULT=A(INTVAL)             ! Call to function AI
RRESULT=A(REALVAL)            ! Call to function AR
RESULT=FUNC(1,MYFUNC)         ! Call to function FUNC1
END PROGRAM MAIN
```

総称インターフェース・ブロックによる組み込みプロシージャーの拡張

総称組み込みプロシージャーは拡張または再定義することが可能です。拡張された組み込みプロシージャーは、既存の特定組み込みプロシージャーを補足します。再定義された組み込みプロシージャーは、既存の特定組み込みプロシージャーと入れ替わります。

総称名と総称組み込みプロシージャー名が同じで、その名前が **INTRINSIC** 属性を持っている場合 (または組み込みコンテキストに指定されている場合)、総称インターフェースは総称組み込みプロシージャーの拡張を行います。

総称名と総称組み込みプロシージャー名は同じでも、その名前が **INTRINSIC** 属性を持っていない場合 (または組み込みコンテキストに指定されていない場合)、総称インターフェースは総称組み込みプロシージャーの再定義を行うことがきます。

総称インターフェース名が **INTRINSIC** 属性を持っている場合 (またはコンテキストに指定されている場合)、総称インターフェース名と特定の組み込みプロシージャー名は同じになることはありません。

組み込みプロシージャーの拡張および再定義の例

```
PROGRAM MAIN
INTRINSIC MAX
INTERFACE MAX                ! Extension to intrinsic MAX
  FUNCTION MAXCHAR(STRING)
    CHARACTER(50) STRING
  END FUNCTION MAXCHAR
END INTERFACE
INTERFACE ABS                ! Redefines generic ABS as
  FUNCTION MYABS(ARG)        !   ABS does not appear in
    REAL(8) MYABS, ARG       !   an INTRINSIC statement
  END FUNCTION MYABS
END INTERFACE
REAL(8) DARG, DANS
REAL(4) RANS
INTEGER IANS,IARG
CHARACTER(50) NAME
DANS = ABS(DARG)             ! Calls external MYABS
IANS = ABS(IARG)             ! Calls intrinsic IABS
```

```
DANS = DABS(DARG)          ! Calls intrinsic DABS
IANS = MAX(NAME)           ! Calls external MAXCHAR
RANS = MAX(1.0,2.0)        ! Calls intrinsic AMAX1
END PROGRAM MAIN
```

定義済み演算子

定義済み演算子とは、ユーザー定義の単項演算子、2 進演算子、または拡張組み込み演算子のことです (122 ページの『拡張組み込みおよび定義済み演算』を参照)。 定義済み演算子は、関数および総称インターフェース・ブロックの両方で定義してください。

1. 単項演算 *op* x_1 は、次のように定義します。
   a. 実際に 1 つの仮引き数 d_1 を指定する関数またはエントリーがなければなりません。
   b. **INTERFACE** ステートメント内の *generic_spec* で、 **OPERATOR** *(op)* を指定します。
   c. x_1 のタイプを、仮引き数 d_1 のタイプと同じにします。
   d. x_1 に型付きパラメーターがある場合は、 d_1 の型付きパラメーターと一致させてください。
   e. 以下を確認します。
      - 関数が **ELEMENTAL** であること。または
      - x_1 が配列の場合、そのランクおよび形状が d_1 と一致していること。
2. 2 進演算 x_1 *op* x_2 は、次のように定義します。
   a. 2 つの仮引き数 d_1 および d_2 を指定する **FUNCTION** または **ENTRY** ステートメントで関数を指定します。
   b. **INTERFACE** 内の *generic_spec* で、 **OPERATOR** *(op)* を指定します。
   c. x_1 および x_2 のタイプを、それぞれ仮引き数 d_1 および d_2 のタイプと同じにします。
   d. x_1 および x_2 に型付きパラメーターがある場合は、それぞれ d_1 および d_2 の型付きパラメーターと一致させてください。
   e. 以下を確認します。
      - 関数が **ELEMENTAL** で、x_1 と x_2 がこれに従ったものになっていること。または
      - x_1 と x_2 のどちらか一方または両方が配列である場合、そのランクと形状がそれぞれ d_1 と d_2 のランクと形状に一致していること。
3. *op* が組み込み演算子である場合、 x_1 または x_2 のいずれかのタイプまたはランクは、組み込み演算子で必須ではありません。
4. *generic_spec* は、引き数のない関数または 2 つ以上の引き数がある関数に対して **OPERATOR** を指定することはできません。
5. 各引き数は必須です。
6. 引き数は **INTENT(IN)** で指定してください。

7. インターフェース・ブロック内で指定されたそれぞれの関数は、想定文字長の結果を保持できません。
8. 指定された演算子が組み込み演算子の場合、その関数の引き数の数はその演算子の組み込み使用数と一致する必要があります。
9. 指定された定義済み演算子は、総称名とともに、複数の関数に適用されます。その場合、それはまさに総称プロシージャー名のような総称となります。組み込み演算子記号の場合、その総称特性には、組み込み演算子記号が表す組み込み演算が含まれます。

IBM 拡張

10. 次の規則は、拡張組み込み演算だけに適用するものです。
    a. 引き数のタイプの 1 つが **BYTE** タイプであるなら、そのとき異なる引き数のタイプは派生型となります。
    b. **-qintlog** コンパイラー・オプションが非文字演算として指定されていて、d_1 が数値または論理である場合、 d_2 は数値または論理にしないでください。
    c. **-qctyplss** コンパイラー・オプションが非文字演算として指定されていて、x_1 が数値または論理で、x_2 が文字定数の場合は、組み込み演算が実行されます。

IBM 拡張 の終り

定義済み演算子の例

```
INTERFACE OPERATOR (.DETERMINANT.)
   FUNCTION IDETERMINANT (ARRAY)
    INTEGER, INTENT(IN), DIMENSION (:,:) :: ARRAY
     INTEGER IDETERMINANT
  END FUNCTION
 END INTERFACE
END
```

定義済み割り当て

定義済み割り当ては、サブルーチンへの参照と見なされます。左側を最初の引き数とし、右側を 2 番目の引き数として括弧でくくります。

1. 定義済み割り当て $x_1 = x_2$ は次のように定義します。
   a. 2 つの仮引き数 d_1 および d_2 を指定する **SUBROUTINE** または **ENTRY** ステートメントでサブルーチンを指定します。
   b. インターフェース・ブロックの *generic_spec* により、 **ASSIGNMENT (=)** を指定します。
   c. x_1 および x_2 のタイプを、それぞれ仮引き数 d_1 および d_2 のタイプと同じにします。
   d. x_1 および x_2 の型付きパラメーターがある場合は、それぞれ d_1 および d_2 の型付きパラメーターと一致させてください。
   e. 以下を確認します。

- サブルーチンが **ELEMENTAL** で、x_1 と x_2 が同じ形状で、x_2 がスカラーであること。または
- x_1 と x_2 のどちらか一方または両方が配列である場合、そのランクと形状をそれぞれ d_1 と d_2 のランクと形状に一致していること。

.

2. **ASSIGNMENT** は、2 つの引き数を持つサブルーチンにのみ使用してください。
3. 各引き数は必須です。
4. 最初の引き数は、**INTENT(OUT)** または **INTENT(INOUT)** のいずれかで、2 番目の引き数は **INTENT(IN)** とすべきです。
5. 引き数のタイプは、両方とも数値、両方とも論理、つまり両方とも同じ種類のパラメーターを持つ文字にしないでください。

IBM 拡張

引き数のタイプの 1 つが **BYTE** タイプであるなら、そのとき異なる引き数のタイプは派生型となります。

-qintlog コンパイラー・オプションが指定されていて、d_1 が数値または論理である場合、 d_2 は数値または論理にしないでください。

-qctyplss コンパイラー・オプションが指定されていて、x_1 が数値または論理で、x_2 が文字定数の場合は、組み込み割り当てが実行されます。

IBM 拡張 の終り

6. **ASSIGNMENT** 総称仕様を指定すると、等号の左右両辺が同じ派生型である場合には、割り当て演算を実行するかまたは再定義します。

定義済み割り当ての例

```
INTERFACE ASSIGNMENT(=)
  SUBROUTINE BIT_TO_NUMERIC (N,B)
    INTEGER, INTENT(OUT) :: N
    LOGICAL, INTENT(IN), DIMENSION(:) :: B
  END SUBROUTINE
END INTERFACE
```

メインプログラム

メインプログラムとは、実行時に実行可能プログラムが呼び出されたときにシステムの制御を受け取るプログラム単位のことです。

```
>>-+-------------------+---------------------------------------><
   '-PROGRAM_statement-'

>>-+--------------------+--------------------------------------><
   '-specification_part-'

>>-+----------------+------------------------------------------><
   '-execution_part-'

>>-+---------------------------+-------------------------------><
   '-internal_subprogram_part-'

>>--END_PROGRAM_statement--------------------------------------><
```

PROGRAM_statement
: 構文の詳細については、 441 ページの『PROGRAM』を参照してください。

specification_part
: 24 ページの『ステートメントおよび実行の順序』で **2** 、 **3** 、および **4** の番号が付けられたステートメント・グループのステートメント・シーケンスです。

execution_part
: 24 ページの『ステートメントおよび実行の順序』 で **3** および **5** の番号が付けられたステートメント・グループのステートメント・シーケンスです。これは、ステートメント・グループ **5** のステートメントで始まらなければなりません。

internal_subprogram_part
: 詳細については、 169 ページの『内部プロシージャー』を参照してください。

END_PROGRAM_statement
: 構文の詳細については、 350 ページの『END』を参照してください。

メインプログラムに、**ENTRY** ステートメントを入れたり、自動オブジェクトを指定したりすることはできません。

IBM 拡張

メインプログラムに **RETURN** ステートメントを入れることはできます。 **RETURN** ステートメントの実行により、**END** ステートメントの実行と同じ効果が得られます。

IBM 拡張 の終り

メインプログラムは、直接、間接を問わずそれ自身を参照することはできません。

モジュール

モジュールには、他のプログラム単位から使用できる仕様および定義が入っています。これらの定義には、データ・オブジェクト定義、名前リスト・グループ、派生型定義、プロシージャーのインターフェース・ブロック、およびプロシージャー定義があります。

IBM 拡張

Fortran 90 モジュールは、グローバル・データを定義します。 **COMMON** データのように、スレッド全体で共用されるため、スレッド・アンセーフとなります。 アプリケーションをスレッド・セーフにするには、グローバル・データを **THREADPRIVATE** または **THREADLOCAL** として宣言する必要があります。詳細については、 314 ページの『COMMON』、 738 ページの『THREADLOCAL』、および 741 ページの『THREADPRIVATE』を参照してください。

IBM 拡張 の終り

```
>>--MODULE_statement-------------------------------------------><

>>--+--------------------+-------------------------------------><
    '-specification_part-'

>>--+------------------------+---------------------------------><
    '-module_subprogram_part-'

>>--END_MODULE_statement---------------------------------------><
```

MODULE_statement
: 構文の詳細については、 416 ページの『MODULE』を参照してください。

specification_part
: 24 ページの『ステートメントおよび実行の順序』で **2**、 **3**、および **4** の番号が付けられたステートメント・グループのステートメント・シーケンスです。

module_subprogram_part

```
>>--CONTAINS_statement----------------------------------------><

   .-----------------.
   V                 |
>>---module_subprogram-+--------------------------------------><
```

CONTAINS_statement
: 構文の詳細については、 324 ページの『CONTAINS』を参照してください。

END_MODULE_statement
: 構文の詳細については、 350 ページの『END』を参照してください。

モジュール・サブプログラムは、モジュール内にありますが、内部サブプログラムではありません。モジュール・サブプログラムは、**CONTAINS** ステートメントの後に続ける必要があり、内部プロシージャーを含めることができます。 モジュール・プロシージャーは、モジュール・サブプログラムまたはモジュール・サブプログラム内のエントリーで定義します。

モジュール内の実行可能ステートメントは、モジュール・サブプログラム内で指定されます。

文字タイプのモジュール関数名の宣言では、アスタリスクを長さ指定として使用することはできません。

specification_part に、ステートメント関数ステートメント、 **ENTRY** ステートメント、または **FORMAT** ステートメントを入れることはできませんが、これらのステートメントをモジュール・サブプログラムの仕様部分に入れることは可能です。

自動オブジェクトおよび **AUTOMATIC** 属性を持つオブジェクトは、モジュールの有効範囲には入れられません。

アクセス可能なモジュール・プロシージャーは、モジュール内の別のサブプログラムまたは使用関連付けを介したモジュールの外側のいずれかの有効範囲単位によって (つまり、**USE** ステートメントを使用することによって) 呼び出すことができます。詳細については、 488 ページの『USE』を参照してください。

IBM 拡張

ポインティング先が定数以外の境界で次元宣言子を指定する場合、整数ポインターを *specification_part* に入れることができません。

モジュールの有効範囲内にあるすべてのオブジェクトは、その関連付け状況、割り振り状況、定義状況、それから使用関連付けを介してモジュールにアクセスするプロシージ

ャーが **RETURN** または **END** ステートメントを実行する際の値を保持します。 詳細については、 74 ページの『未定義を発生させるイベント』の項目 4 を参照してください。

IBM 拡張 の終り

モジュールはモジュール・プロシージャーまたは派生型定義に対するホストとなり、ホスト関連付けを介してモジュールの有効範囲内のエンティティーにアクセスできます。

モジュール・プロシージャーは、仮プロシージャー引き数に関連した実引き数として使用することができます。

モジュール・プロシージャーの名前は、モジュールの有効範囲に対してローカルで、共通ブロック名以外のモジュール内のどのエンティティー名とも同じにすることはできません。

マイグレーションのためのヒント:

- 共通ブロックおよび **INCLUDE** ディレクティブを除去します。
- モジュールを使用してグローバル・データおよびプロシージャーを保持し、定義の整合性が取れるようにします。

FORTRAN 77 ソース

```
        COMMON /BLOCK/A, B, C,
NAME, NUMBER
        REAL A, B, C
        A = 3
        CALL CALLUP(D)
        PRINT *, NAME, NUMBER
        END
        SUBROUTINE CALLUP (PARM)
          COMMON /BLOCK/A, B, C, NAME, NUMBER
          REAL A, B, C
          ...
          NAME = 3
          NUMBER = 4
        END
```

Fortran 90 または Fortran 95 ソース

```
        MODULE
FUNCS
          REAL A, B, C            ! Common block no longer needed
          INTEGER NAME, NUMBER    ! Global data
          CONTAINS
             SUBROUTINE CALLUP (PARM)
               ...
               NAME = 3
               NUMBER = 4
             END SUBROUTINE
        END MODULE FUNCS
        PROGRAM MAIN
        USE FUNCS
        A = 3
        CALL CALLUP(D)
        PRINT *, NAME, NUMBER
        END
```

モジュールの例

```
MODULE M
  INTEGER SOME_DATA
  CONTAINS
    SUBROUTINE SUB()                        ! Module subprogram
      INTEGER STMTFNC
      STMTFNC(I) = I + 1
      SOME_DATA = STMTFNC(5) + INNER(3)
```

```
      CONTAINS
        INTEGER FUNCTION INNER(IARG)      ! Internal subprogram
          INNER = IARG * 2
        END FUNCTION
    END SUBROUTINE SUB
END MODULE
PROGRAM MAIN
  USE M                                   ! Main program accesses
  CALL SUB()                              !   module M
END PROGRAM
```

ブロック・データのプログラム単位

ブロック・データのプログラム単位は、名前付き共通ブロック内のオブジェクトに初期値を与えます。

```
>>--BLOCK_DATA_statement------------------------------------------><

>>--+--------------------+----------------------------------------><
    '-specification_part-'

>>--END_BLOCK_DATA_statement--------------------------------------><
```

BLOCK_DATA_statement
: 構文の詳細については、 298 ページの『BLOCK DATA』を参照してください。

specification_part
: 24 ページの『ステートメントおよび実行の順序』で **2**、 **3**、および **4** の番号が付けられたステートメント・グループのステートメント・シーケンスです。

END_BLOCK_DATA_statement
: 構文の詳細については、 350 ページの『END』を参照してください。

specification_part では、タイプ宣言、**USE**、 **IMPLICIT**、**COMMON**、**DATA**、**EQUIVALENCE**、および整数 **POINTER** ステートメント、派生型定義、および使用可能な属性仕様ステートメントを指定することができます。指定可能な属性としては、**PARAMETER**、**DIMENSION**、 **INTRINSIC**、**POINTER**、**SAVE**、および **TARGET** があります。

ブロック・データ *specification-part* 内のタイプ宣言ステートメントには、ALLOCATABLE または EXTERNAL 属性指定子を含んではなりません。

1 つの実行可能プログラム内に複数のブロック・データ・プログラム単位を入れることができますが、名前を指定しなくてよいのはその中の 1 つだけです。 また、ブロック・データ・プログラム単位内の複数の名前付き共通ブロックを初期化することができます。

ブロック・データ・プログラム単位内の共通ブロックに関する制約事項は、次のとおりです。
- 名前付き共通ブロック内のすべての項目は、その初期化がすべて終了していなくても、 **COMMON** ステートメント内に入っている必要があります。
- 同一の名前付き共通ブロックを、2 つの異なるブロック・データ・プログラム単位内で参照することはできません。
- ブロック・データ・プログラム単位内で初期化できるのは、名前付き共通ブロック内のポインター以外のオブジェクトのみとなります。
- ブランクの共通ブロック内のオブジェクトは初期化されません。

ブロック・データ・プログラム単位の例

```
PROGRAM MAIN
  COMMON /L3/ C, X(10)
  COMMON /L4/ Y(5)
END PROGRAM
BLOCK DATA BDATA
  COMMON /L3/ C, X(10)
  DATA C, X /1.0, 10*2.0/   ! Initializing common block L3
END BLOCK DATA

BLOCK DATA                  ! An unnamed block data program unit
  PARAMETER (Z=10)
  DIMENSION Y(5)
  COMMON /L4/ Y
  DATA Y /5*Z/
END BLOCK DATA
```

関数およびサブルーチン・サブプログラム

サブプログラムは、関数またはサブルーチンのどちらかであり、内部サブプログラム、外部サブプログラム、モジュール・サブプログラムのいずれかです。また、ステートメント関数のステートメントに関数を指定することもできます。外部サブプログラムは一種のプログラム単位です。

```
►►—subprogram_statement—————————————————————————————————————►◄

►►——┬———————————————————┬———————————————————————————————————►◄
    └—specification_part—┘

►►——┬———————————————┬———————————————————————————————————————►◄
    └—execution_part—┘

►►——┬—————————————————————————┬—————————————————————————————►◄
    └—internal_subprogram_part—┘

►►—end_subprogram_statement—————————————————————————————————►◄
```

subprogram_statement
: 構文の詳細については、 378 ページの『FUNCTION』または 473 ページの『SUBROUTINE』を参照してください。

specification_part
: 24 ページの『ステートメントおよび実行の順序』で **2**、 **3**、および **4** の番号が付けられたステートメント・グループのステートメント・シーケンスです。

execution_part
: 24 ページの『ステートメントおよび実行の順序』 で **3** および **5** の番号が付けられたステートメント・グループのステートメント・シーケンスです。これは、ステートメント・グループ **5** のステートメントで始まらなければなりません。

internal_subprogram_part
: 詳細については、 169 ページの『内部プロシージャー』を参照してください。

end_subprogram_statement
: 関数およびサブルーチンの **END** ステートメントに関する詳細については、 350 ページの『END』を参照してください。

内部サブプログラムは、メインプログラム、モジュール・サブプログラム、または外部サブプログラムにある **CONTAINS** ステートメントの*後* で、そしてホスト・プログラムの **END** ステートメントの*前* で宣言します。内部サブプログラムの名前は、ホスト有効範囲単位内の仕様セクションには定義しないでください。

外部プロシージャーは、実行可能プログラムに関してグローバルな有効範囲を持っています。呼び出し側プログラム単位では、インターフェース・ブロック内の外部プロシージャーにインターフェースを指定したり、 **EXTERNAL** 属性で外部プロシージャー名を定義することができます。

サブプログラムには、**PROGRAM**、 **BLOCK DATA**、 **MODULE** 以外のステートメントであれば、どのステートメントを指定してもかまいません。 内部サブプログラムには、 **ENTRY** ステートメントや内部サブプログラムを指定することはできません。

プロシージャー参照

プロシージャー参照には、次の 2 つのタイプがあります。

- サブルーチンは、**CALL** ステートメント (詳細については、 302 ページの『CALL』を参照) または定義済み割り当てステートメントの実行で呼び出します。
- 関数は、関数参照または定義済み演算の実行時に呼び出します。

関数参照

関数参照は、次のように式の中の 1 次子として使用します。

```
►►─function_name─(──────────────────────────)─────────────────────────►◄
                   └─actual_argument_spec_list─┘
```

関数参照が実行されると、次のことが順番に行われます。

1. 実引き数になる式が評価されます。
2. 実引き数が、対応する仮引き数に関連付けられます。
3. 制御が指定された関数に移ります。
4. 関数が実行されます。
5. 関数の結果変数値 (ポインター関数の場合は、状況またはターゲット) が、参照している式で使えるようになります。

関数参照の実行によって、その関数参照が入っているステートメント内部の他のデータ項目値を変更しないでください。 論理 **IF** ステートメント または **WHERE** ステートメントの論理式にある関数参照の起動により、その式の値が真である場合に実行されるステートメントのエンティティーに影響を与える可能性があります。

IBM 拡張

値および参照により引き数を引き渡すことによって言語間呼び出しを容易にするために、それぞれの場合に使用するための引き数リスト組み込み関数 **%VAL** および **%REF** が提供されています。これらの組み込み関数は、Fortran 以外のプロシージャー参照およびインターフェース本体のサブプログラム・ステートメントで指定できます。 (197 ページの『%VAL および %REF』を参照してください。) 参照関数の例については 関数ステートメントおよび 再帰を参照してください。

IBM 拡張 の終り

割り振り可能な関数には、明示的インターフェースがあります。

割り振り可能な関数に入ると、結果変数の割り振り状況は、現在割り振られていない状態になります。

関数の結果変数は、関数の実行中に何度でも割り振りまたは割り振り解除を行うことができます。ただし、それが現在割り振り済みで、関数を終了するときに定義済みの値がある必要があります。結果変数の自動割り振り解除は、関数を終了するときには即時には行われません。その代わりに、関数の参照が発生するステートメントの実行後に行われます。

サブプログラムおよびプロシージャー参照の例

```
PROGRAM MAIN
REAL QUAD,X2,X1,X0,A,C3
QUAD=0; A=X1*X2
X2 = 2.0
X1 = SIN(4.5)                    ! Reference to intrinsic function
X0 = 1.0
CALL Q(X2,X1,X0,QUAD)            ! Reference to external subroutine
C3 = CUBE()                      ! Reference to internal function
CONTAINS
  REAL FUNCTION CUBE()           ! Internal function
    CUBE = A**3
  END FUNCTION CUBE
END
SUBROUTINE Q(A,B,C,QUAD)         ! External subroutine
  REAL A,B,C,QUAD
  QUAD = (-B + SQRT(B**2-4*A*C)) / (2*A)
END SUBROUTINE Q
```

割り振り可能な関数の結果の例

```
FUNCTION INQUIRE_FILES_OPEN() RESULT(OPENED_STATUS)
  LOGICAL,ALLOCATABLE :: OPENED_STATUS(:)
  INTEGER I,J
  LOGICAL TEST
  DO I=1000,0,-1
    INQUIRE(UNIT=I,OPENED=TEST,ERR=100)
    IF (TEST) EXIT
100 CONTINUE
  END DO
  ALLOCATE(OPENED_STATUS(0:I))
  DO J=0,I
    INQUIRE(UNIT=J,OPENED=OPENED_STATUS(J))
  END DO
END FUNCTION INQUIRE_FILES_OPEN
```

組み込みプロシージャー

組み込みプロシージャーは、XL Fortran によってすでに定義されているプロシージャーです。詳細については、539 ページの『第 12 章 組み込みプロシージャー』を参照してください。

組み込みプロシージャーには、総称名で参照できるもの、特定名で参照できるもの、および両方で参照できるものがあります。

総称組み込み関数
: では、いくつかの例外を除いて、引き数が特定のタイプである必要はなく、結果はその引き数のタイプと同じになるのが普通です。同じプロシージャー名を複数のタイプの引き数 (タイプおよび kind 型付きパラメーターの引き数はどの特定の関数を使用するかを判別します) を指定して使用することにより、総称名は簡単に組み込みプロシージャーを参照することができます。

特定組み込み関数
: では、特定のタイプの引き数を必要とし、結果も特有のタイプとなります。

  特定組み込み関数名は、実引き数として渡すことができます。特定組み込み関数の名前が総称組み込み関数と同じ場合、特定名が参照されます。特定組み込みプロシージャーに関連している仮プロシージャーに対するすべての参照においては、その組み込みプロシージャーのインターフェースと整合性のある引き数を使用する必要があります。

組み込みプロシージャーの名前を引き数として渡せるか否かについては、プロシージャーによって異なります。 プロシージャー参照において **INTRINSIC** 属性を実引き数として指定されている組み込みプロシージャーの特定名を使用することができます。

- **IMPLICIT** ステートメントは、組み込み関数のタイプを変更しません。
- 組み込み名が **INTRINSIC** 属性で指定される場合、その名前は必ず組み込みプロシージャーとして認識されます。

組み込みプロシージャー名と他の名前の不一致

組み込みプロシージャー名は認識されるので、データ・オブジェクトが組み込みプロシージャーと同じ名前で宣言されていると、その組み込みプロシージャーは使用することができません。

総称インターフェース・ブロックは、 172 ページの『インターフェース・ブロック』に記述されているように、総称組み込み関数の拡張または再定義を行うことができます。その関数がすでに **INTRINSIC** 属性を持っている場合は拡張され、持っていない場合は再定義されます。

引き数

実引き数の仕様

```
►►─┬───────────────┬─┬─argument──────────────┬────────────────────►◄
   └─arg_keyword─ = ┘ │               (1)     │
                     ├─%VAL─(─argument──────)─┤
                     │               (2)     │
                     └─%REF─(─argument──────)─┘
```

注:

1　IBM 拡張

2　IBM 拡張

arg_keyword
　呼び出し中のプロシージャーの明示的インターフェース内の仮引き数名です。

argument
　実引き数です。

IBM 拡張

%VAL、%REF
　引き渡し方法を指定します。詳細については、 197 ページの『%VAL および %REF』を参照してください。

IBM 拡張 の終り

実引き数は、プロシージャー参照の引き数リストに入れます。プロシージャー参照の実引き数は下記のいずれかです。
- 式
- 変数
- プロシージャー名
- 選択戻り指定子 (実引き数が **CALL** ステートメントにある場合)。 **stmt_label* 形式になります。*stmt_label* は、 **CALL** ステートメントと同じ有効範囲単位にある分岐ターゲット・ステートメントのステートメント・ラベルを表しています。

ステートメント関数参照内で指定される実引き数は、スカラー・オブジェクトである必要があります。

プロシージャー名は、特定名でない場合、内部プロシージャーの名前、ステートメント関数、またはプロシージャーの総称名にすることはできません。

プロシージャーの参照の規則および制約事項は、 190 ページの『プロシージャー参照』で説明しています。 F95 Fortran 95 では、非組み込みエレメント型プロシージャーを実引き数として使用することはできません。 F95

引き数キーワード

引き数キーワードを使用して、実引き数を仮引き数とは異なる順番で指定することができます。引き数キーワードを使用すると、オプションの仮引き数に対応する実引き数を省略することができます。つまり、単にプレースホルダーとして機能する仮引き数を必要としません。

各引き数キーワードは、参照中のプロシージャーの明示的インターフェースにある仮引き数の名前にしてください。引き数キーワードは、暗黙のインターフェースを持つプロシージャーの引き数リスト内に入れないでください。

引き数リスト内で、実引き数を引き数キーワードを使用して指定される場合、リストの次の実引き数も引き数キーワードによって指定されます。

引き数キーワードは、ラベル・パラメーターに対して指定することはできません。ラベル・パラメーターは、そのプロシージャー参照内で引き数キーワードを参照する前に指定しなければなりません。

引き数キーワードの例:

```
INTEGER MYARRAY(1:10)
INTERFACE
  SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
    INTEGER ARRAY_SIZE, ARRAY(ARRAY_SIZE)
    LOGICAL, OPTIONAL :: DESCENDING
  END SUBROUTINE
END INTERFACE
CALL SORT(MYARRAY, ARRAY_SIZE=10)  ! No actual argument corresponds to the
                                   ! optional dummy argument DESCENDING
END
SUBROUTINE SORT(ARRAY, DESCENDING, ARRAY_SIZE)
  INTEGER ARRAY_SIZE, ARRAY(ARRAY_SIZE)
  LOGICAL, OPTIONAL :: DESCENDING
  IF (PRESENT(DESCENDING)) THEN
            .
            .
            .
END SUBROUTINE
```

仮引き数

```
►►─┬─dummy_arg_name──────────────────────┬──────────────────────────────────►◄
   │      (1)                            │
   ├─%VAL────────(──dummy_arg_name──)─┤
   │      (2)                            │
   └─%REF────────(──dummy_arg_name──)─┘
```

注:

1 IBM 拡張

2 IBM 拡張

仮引き数は、ステートメント関数ステートメント、**FUNCTION** ステートメント、**SUBROUTINE** ステートメント、または **ENTRY** ステートメントで指定します。ステートメント関数、関数サブプログラム、インターフェース本体、およびサブルーチン・サブプログラムの仮引き数は、実引き数のタイプや、および各引き数がスカラー値、配列、プロシージャー、またはステートメント・ラベルのいずれであるかを示しています。外部サブプログラム、モジュール・サブプログラム、内部サブプログラムの定義内、またはインターフェース本体内の仮引き数は、次のいずれかに分類されます。
- 変数名
- プロシージャー名
- アスタリスク (サブルーチン内専用、選択戻りポイントを示します)

IBM 拡張

%VAL または **%REF** は、インターフェース・ブロックにある **FUNCTION** ステートメントまたは **SUBROUTINE** ステートメントの仮引き数に対してのみ指定することができます。インターフェースは、Fortran 以外のプロシージャー・インターフェースに合ったものにしてください。 **%VAL** または **%REF** が外部プロシージャーのインターフェース・ブロックにある場合、この引き渡し方法はそのプロシージャーに対する参照ごとに示されます。外部プロシージャーの参照で実引き数が **%VAL** または **%REF** を指定する場合、対応する仮引き数用のインターフェース・ブロックで同じ引き渡し方法を指定する必要があります。詳細については、 197 ページの『%VAL および %REF』を参照してください。

IBM 拡張 の終り

ステートメント関数定義の仮引き数は、変数名として分類されます。

指定名は、仮引き数リスト内に一度だけ入れることが可能です。

ステートメント関数のステートメントに仮引き数として指定される変数名の有効範囲は、指定されているステートメントの有効範囲と同じです。この変数名のタイプは、有

効範囲単位内のステートメント関数を含む変数名である場合に持つタイプおよび型付きパラメーターと同じです。変数名は、アクセス可能な配列と同じ名前にはできません。

引き数関連付け

実引き数は、関数またはサブルーチンが参照されると、仮引き数に関連付けられます。プロシージャー参照では、実引き数リストが、そのリストにある実引き数とサブプログラムの仮引き数との間の対応を識別します。

引き数キーワードがない場合、実引き数は仮引き数リスト内で対応する位置を占める仮引き数に関連付けられます。最初の実引き数が最初の仮引き数と、2 番目の実引き数が 2 番目の仮引き数といったように、以下同様に関連付けられます。各実引き数は、仮引き数に関連付ける必要があります。

キーワードがある場合、実引き数は引き数キーワードと同じ名前の仮引き数に関連付けられます。プロシージャー参照が入っている有効範囲単位において、仮引き数の名前はアクセス可能な明示的インターフェース内にある必要があります。

サブプログラム内の引き数関連付けは、そのプログラム内で **RETURN** または **END** ステートメント実行時に終了します。 サブプログラムが参照されたときの引き数関連付けが、次のサブプログラムの参照時まで持ち越されることはありません。ただし、**-qxlf77** コンパイラー・オプションの **persistent** サブオプションが指定され、サブプログラムに 1 つ以上のエントリー・プロシージャーが指定されている場合には、引き数関連付けは次の参照時まで持ち越されます。

IBM 拡張

%VAL が使用される場合を除いて、サブプログラムでは仮引き数のためにストレージが予約されることはありません。サブプログラムの計算用には、対応する実引き数が使用されます。したがって、仮引き数が変わると、実引き数の値も変わります。対応する実引き数が式またはベクトル添え字付きの配列セクションである場合、呼び出し側のプロシージャーは実引き数のためにストレージを予約し、サブプログラムは仮引き数を定義、再定義、または定義解除することはできません。

実引き数が **%VAL** で指定されるか、または対応する仮引き数に **VALUE** 属性がある場合、サブプログラムには、実引き数のストレージ域へのアクセス権がありません。

IBM 拡張 の終り

実引き数は、対応する仮引き数で指定したタイプおよび型付きパラメーター (仮引き数がポインターまたは想定形状のいずれかの場合は、形状) と一致することが必要です。ただし、サブルーチン名にタイプがなく、サブルーチンである仮プロシージャー名と関連付ける必要がある場合、および選択戻り指定子にタイプがなく、アスタリスクと関連付ける必要がある場合を除きます。

引き数関連付けは、複数レベルにわたるプロシージャー参照まで、持ち越される場合があります。

サブプログラムの参照によって、参照されたサブプログラム内の仮引き数が、参照されたサブプログラム内の別の仮引き数に関連付けられる場合、そのサブプログラムの実行時にどちらの仮引き数も定義、再定義、または未定義状態にすることはできません。たとえば、次のようなサブルーチンの定義があるとします。

```
SUBROUTINE XYZ (A,B)
```

このサブルーチンを、次の方法で参照すると、

```
CALL XYZ (C,C)
```

仮引き数 A および B がそれぞれ同じ引き数 C に関連付けられます。その結果、A と B が互いに関連することになります。この場合、A も B も、サブルーチン XYZ の実行時または XYZ によって参照されるいずれのプロシージャーによっても定義、再定義、または未定義状態にすることはできません。

共通ブロック内のエンティティーまたは使用関連付けやホスト関連付けを介してアクセス可能なエンティティーによって、仮引き数が関連する場合、エンティティーが仮引き数に関連している間に、エンティティーの値は仮引き数名の使用によってのみ変更することができます。データ・オブジェクトのいずれかの部分が仮引き数によって定義される場合、そのデータ・オブジェクトは、定義が発生する前か後かにかかわらず、その仮引き数によってのみ参照することができます。この制限は、ポインター・ターゲットについても適用されます。

IBM 拡張

この制限に従わないプログラムがある場合、コンパイラー・オプション **-qalias=nostd** を使用するのが適切です。詳細については、「ユーザーズ・ガイド」の『**-qalias** オプション』を参照してください。

IBM 拡張 の終り

%VAL および %REF

IBM 拡張

Fortran 以外の言語で書かれたサブプログラム (たとえば、ユーザー作成の C プログラム、 AIX オペレーティング・システム・ルーチン) を呼び出すには、 XL Fortran で使用されているデフォルトの方法とは異なる方法で実引き数を渡さなければならない場合があります。デフォルトの方法では、実引き数のアドレスを渡し、実引き数が文字タイプの場合は長さを渡します。 (**-qnullterm** コンパイラー・オプションを使用して、スカ

ラー文字初期化式が終端ヌル・ストリングとともに渡されるようにしてください。詳細については、「ユーザーズ・ガイド」の **-qnullterm** を参照してください。)

CALL ステートメントまたは関数参照ステートメントの引き数リスト内で、**%VAL** および **%REF** 組み込み関数を使用することによって、またはインターフェース本体内で仮引き数を使うことによって、デフォルトの引き渡し方法を変更することができます。これらの組み込み関数は、外部サブプログラムに実引き数を渡す方法を指定します。

%VAL および **%REF** 組み込み関数は、Fortran プロシージャー参照の引き数リスト内で使用することはできません。また、選択戻り指定子とともに使用することもできません。

引き数リストの組み込み関数には、次のものがあります。

%VAL この組み込み関数は、**CHARACTER(1)**、論理、整数、実数、複素数の式、または順序列の派生型である実引き数と一緒に使用することができます。派生型のオブジェクトは、長さが 1 バイトより長い文字構造体コンポーネントや配列を持つことはできません。

%VAL は、配列、プロシージャー名、または 1 バイトより長い文字式である実引き数と一緒に使用することはできません。

%VAL が実行されると、実引き数が、32 ビットまたは 64 ビットの中間値として渡されます。実引き数が実数タイプまたは複素数タイプである場合、その実引き数は 1 つまたは複数の 64 ビット中間値として渡されます。実引き数が整数タイプ、論理タイプ、または順序派生型である場合、その実引き数は 1 つ以上の 32 ビット中間値として渡されます。 32 ビットより短い整数実引き数は、32 ビット値になるまで符号が拡張され、 32 ビットより短い論理実引き数は、32 ビット値になるまでゼロで埋められます。

名前付きバイト定数および変数は、**INTEGER(1)** であるかのように渡されます。実引き数が **CHARACTER(1)** であるとき、 **-qctyplss** コンパイラー・オプションが指定されたか否かにかかわらず、その左側が 32 ビット値になるまでゼロで埋められます。

%REF この組み込み関数を実行すると、実引き数が渡され、参照できるようになります。実際には、実引き数のアドレスだけが渡されます。デフォルトの引き渡し方法とは異なり、**%REF** は文字引き数の長さを渡しません。このような文字引き数を C ルーチンに渡す場合は、 C ルーチンがストリングの長さを判別できるように、(たとえば、 **-qnullterm** オプションを使って) ストリングをヌル文字で終了させる必要があります。

%VAL および %REF の例

```
EXTERNAL FUNC
CALL RIGHT2(%REF(FUNC))       ! procedure name passed by reference
REAL XVAR
CALL RIGHT3(%VAL(XVAR))       ! real argument passed by value
```

```
IVARB=6
CALL TPROG(%VAL(IVARB))       ! integer argument passed by value
```

%VAL の標準準拠代替については、 491 ページの『VALUE』を参照してください。

詳細については、「ユーザーズ・ガイド」の『*言語間呼び出し*』を参照してください。

IBM 拡張 の終り

仮引き数の意図

INTENT 属性により、仮引き数の意図的な使用を明示的に指定することができます。明示的インターフェースがある場合、この属性を使用してプログラムの呼び出しプロシージャーの最適化を改善させることもできます。また、引き数の意図が明らかになることにより、エラーを検査する機会が多くなります。構文の詳細については、 405 ページの『INTENT』を参照してください。

IBM 拡張

次の表は、内部プロシージャーについて XL Fortran の引き数を渡す方法を概略します (想定形状仮引き数およびポインター仮引き数は含まれません)。

表 5. 引き渡し方法および意図

引き数タイプ	Intent(IN)	Intent(OUT)	Intent(INOUT)	No Intent
非 CHARACTER Scalar	**VALUE**	デフォルト	デフォルト	デフォルト
CHARACTER*1 Scalar	**VALUE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*n Scalar	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*(*) Scalar	デフォルト	デフォルト	デフォルト	デフォルト
派生型 [1] Scalar	**VALUE**	デフォルト	デフォルト	デフォルト
派生型 [2] Scalar	デフォルト	デフォルト	デフォルト	デフォルト
非 CHARACTER Array	デフォルト	デフォルト	デフォルト	デフォルト
CHARACTER*1 Array	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**
CHARACTER*n Array	**REFERENCE**	**REFERENCE**	**REFERENCE**	**REFERENCE**

表 5. 引き渡し方法および意図 (続き)

引き数タイプ	Intent(IN)	Intent(OUT)	Intent(INOUT)	No Intent
CHARACTER*(*) Array	デフォルト	デフォルト	デフォルト	デフォルト
派生型 [3] Array	デフォルト	デフォルト	デフォルト	デフォルト

IBM 拡張 の終り

オプションの仮引き数

OPTIONAL 属性を指定すると、プロシージャーへの参照で、仮引き数を実引き数に関連付ける必要はなくなります。 **OPTIONAL** 属性の利点には、次のようなものがあります。

- オプションの仮引き数を使用して、デフォルトの動作をオーバーライドします。たとえば、194 ページの『引き数キーワードの例』を参照してください。
- より柔軟にプロシージャーを参照できます。たとえば、プロシージャーにエラー・ハンドラーまたは戻りコード用のオプションの引き数を指定できます。しかし、どのプロシージャー参照が対応する実引き数を提供するかを選択することができます。

構文および規則に関する詳細については、427 ページの『OPTIONAL』を参照してください。

指定されていないオプションの仮引き数に対する制限事項

ある仮引き数が実引き数と関連しており、かつ、この実引き数が呼び出しサブプログラム内にオプションでない仮引き数として指定されている場合、または呼び出しサブプログラム内に仮引き数として指定されていない場合、その仮引き数はサブプログラムのインスタンス内に指定されています。仮引き数のうちオプションでないものは、必ず指定しなければなりません。

オプションの仮引き数のうち指定されていないものは、以下の規則に従う必要があります。

- 仮データ・オブジェクトの場合、定義または参照はできません。仮データ・オブジェクトがデフォルトの初期化が指定できるタイプである場合、その初期化には効果はありません。
- 仮プロシージャーの場合、呼び出しはできません。

1. 配列コンポーネントまたは CHARACTER*n コンポーネント ($n > 1$) を持たない派生型のデータ・オブジェクト

2. 配列コンポーネントまたは CHARACTER*n コンポーネント ($n > 1$) を持つ派生型のデータ・オブジェクト

3. 任意のタイプ、サイズ、ランクのコンポーネントを持つ派生型のデータ・オブジェクト

- オプションではない仮引き数に対応する実引き数として指定することはできません。ただし、**PRESENT** 組み込み関数の引き数として指定することはできます。
- オプションの仮引き数のサブオブジェクトのうち指定されていないものは、オプションの仮引き数に対応する実引き数として使用することはできません。
- 指定されていないオプションの仮引き数が配列である場合、実引き数としてエレメント型プロシージャーに提供することはできません。ただし、同じランクの配列が、そのエレメント型プロシージャーのオプションでない仮引き数に対応する実引き数として提供される場合を除きます。
- 指定されていないオプションの仮引き数がポインターである場合、オプションではない仮引き数に対応する実引き数として指定することはできません。ただし、**PRESENT** 組み込み関数の引き数として指定することはできます。
- 存在しないオプションの仮引き数が割り振り可能である場合、割り振り可能ではない仮引き数に対応する実引き数として、割り振り、割り振り解除、または提供を行うことはできません。ただし、**PRESENT** 組み込み関数の引き数としてこれを行うことはできます。

文字引き数の長さ

文字仮引き数の長さが非定数の宣言式と同じである場合、オブジェクトは実行時の長さを持つ仮引き数です。仮引き数ではないオブジェクトが実行時の長さを持つ場合、自動オブジェクトです。詳細については、 29 ページの『自動オブジェクト』を参照してください。

仮引き数の長さ指定子が括弧で囲まれたアスタリスクの場合、仮引き数の長さは実引き数から「継承され」ます。仮引き数を含んでいるプログラム単位外で長さが指定されているため、その長さを継承することになるわけです。関連した実引き数が配列名である場合、仮引き数が継承する長さは、関連した実引き数配列における配列エレメントの長さです。継承された長さで **%REF** を文字仮引き数に指定することはできません。

仮引き数としての変数

変数である仮引き数は、同じタイプおよび kind 型付きパラメーターを持つ変数である実引き数に関連付ける必要があります。

実引き数がスカラーである場合、対応する仮引き数もスカラーでなければなりません。ただし、実引き数が想定形状配列またはポインター配列 (あるいはこのようなエレメントのサブストリング) でない配列のエレメントの場合を除きます。実引き数が割り振り可能な場合は、対応する仮引き数も割り振り可能でなければなりません。プロシージャーが総称名によって参照されるか、定義済み演算子または定義済み割り当てとして参照される場合、実引き数および対応する仮引き数のランクは一致する必要があります。スカラー仮引き数は、スカラー実引き数にのみ関連付けることができます。

Fortran 95

エレメント型サブプログラム内で使用される仮引き数には以下が適用されます。

- すべての仮引き数はスカラーでなければならず、**POINTER** または **ALLOCATABLE** 属性を持つことはできません。
- 仮引き数、またはそのサブオブジェクトは、宣言式内で使用することはできません。ただし、それが **BIT_SIZE**、**KIND**、または **LEN** 組み込み関数への引き数として、または数値照会の組み込み関数のいずれかへの引き数として使用される場合を除きます。これについては 539 ページの『第 12 章 組み込みプロシージャー』を参照してください。
- 仮引き数にアスタリスクを指定することはできません。
- 仮引き数に仮プロシージャーを指定することはできません。

Fortran 95 の終り

スカラー仮引き数が文字タイプである場合、その長さは実引き数の長さ以下になります。仮引き数は、実引き数の左端の文字に関連付けられます。文字タイプの仮引き数が配列である場合、長さの制限は各配列エレメントにではなく配列全体に適用されます。つまり、仮引き数配列全体の長さを実引き数配列全体より長くすることはできませんが、関連した配列エレメントの長さを変更することはできます。

仮引き数が想定形状配列である場合、実引き数は想定サイズ配列またはスカラー (配列エレメントまたは配列エレメント・サブストリング用の指定子を含む) にしてはなりません。

仮引き数が明示的形状配列または想定サイズ配列で、実引き数が文字以外の配列である場合、仮引き数のサイズは実引き数配列のサイズを超えてはなりません。各実配列エレメントは、対応する仮配列エレメントに関連させられます。実引き数が `as` という添え字値を持つ文字以外の配列エレメントである場合、仮引き数配列のサイズは、実引き数配列のサイズ `+ 1 - as` を超えてはなりません。 `ds` という添え字値を持つ仮引き数配列エレメントは、`as + ds - 1` という添え字値を持つ実引き数配列エレメントに関連付けられます。

実引き数が文字配列、文字配列エレメント、または文字サブストリングのいずれかで、配列の文字記憶単位 `acu` で始まる場合、関連した仮引き数配列の文字記憶単位 `dcu` は、実配列引き数の文字記憶単位 `acu+dcu-1` に関連付けられます。

関連した実引き数が変数の場合、変数名である仮引き数をサブプログラム内で定義することができます。関連した実引き数が定義可能ではない場合、変数名である仮引き数をサブプログラム内で再定義してはなりません。

実引き数がベクトル添え字を持つ配列セクションの場合、関連付けられる仮引き数を定義することができません。

ポインター以外の仮引き数がポインター実引き数に関連付けられる場合、その実引き数は、仮引き数が引き数と関連するターゲットに現在関連付けられている必要があります。実引き数を渡す方法に関する制限はすべて、実引き数のターゲットに適用されます。

仮引き数がターゲットでもポインターでもない場合は、実引き数に関連したポインターは、プロシージャー呼び出し時に対応する仮引き数に関連付けられません。

仮引き数と実引き数の両方がターゲットであり、その仮引き数がスカラーまたは想定形状配列の場合 (しかも、実引き数がベクトル添え字を持つ配列セクションでない場合) は、次のようになります。

1. 実引き数に関連したポインターはすべて、プロシージャー呼び出し時に対応する仮引き数に関連付けられます。
2. プロシージャーの実行が完了しても、仮引き数に関連したポインターは実引き数に関連付けられたままになります。

仮引き数と実引き数の両方がターゲットであり、その仮引き数が明示的形状配列または想定サイズ配列のいずれかで、一方の実引き数がベクトル添え字を持つ配列セクションでない場合は、次のようになります。

1. 実引き数に関連付けられたポインターが、プロシージャーの呼び出し時に対応する仮引き数に関連付けられるかどうかは、プロセッサー次第です。
2. プロシージャーの実行が完了しても、仮引き数に関連付けられたポインターが実引き数に関連付けられたままになるかどうかは、プロセッサー次第です。

仮引き数がターゲットであり、それに対応する実引き数がターゲットではないか、またはベクトル添え字を持つ配列セクションである場合、仮引き数に関連付けられたポインターは、プロシージャーの実行完了時に未定義になります。

仮引き数として割り振り可能なオブジェクト

割り振り可能な仮引き数は、それに関連付けられた、割り振り可能な実引き数を持ちます。割り振り可能な仮引き数が配列の場合、関連付けられた実引き数も配列でなければなりません。

プロシージャーに入るときには、割り振り可能な仮引き数の割り振り状況は、関連付けられた実引き数の割り振り状況になります。仮引き数が INTENT(OUT) で、関連付けられた実引き数が現在割り振られている場合は、仮引き数の割り振り状況が、現在割り振られていないという割り振り状況になるように、プロシージャー呼び出しでその実引き数が割り振り解除されます。仮引き数が INTENT(OUT) ではなく、実引き数が現在割り振られている場合は、仮引き数の値は関連付けられた実引き数の値になります。

プロシージャーがアクティブの間に、INTENT(IN) を持たない割り振り可能な仮引き数の割り振り、割り振り解除、定義、または定義解除を行うことができます。これらのイベントのいずれかが発生した場合、別名を介して関連した実引き数を参照することは許されません。

ルーチンの終了時には、実引き数は割り振り可能な仮引き数の割り振り状況を持ちます (割り振り可能な仮引き数が INTENT(IN) を持つ場合も、当然、変更はありません)。仮引き数から実引き数への値の伝搬のために、通常の規則が適用されます。

割り振り可能な仮引き数の自動割り振り解除は、仮引き数のプロシージャー内の RETURN または END ステートメントの実行結果として発生しません。

注: INTENT(IN) 割り振り可能仮引き数は、呼び出し先のプロシージャー内で変更された、割り振り状況を持つことはできません。これらの仮引き数と、通常の仮引き数との主な違いは、プロシージャーに入るとき (およびプロシージャーの実行中) に割り振り解除できるということです。

例

```
SUBROUTINE LOAD(ARRAY, FILE)
   REAL, ALLOCATABLE, INTENT(OUT) :: ARRAY(:, :, :)
   CHARACTER(LEN=*), INTENT(IN) :: FILE
   INTEGER UNIT, N1, N2, N3
   INTEGER, EXTERNAL :: GET_LUN
   UNIT = GET_LUN() ! Returns an unused unit number
   OPEN(UNIT, FILE=FILE, FORM='UNFORMATTED')
   READ(UNIT) N1, N2, N3
   ALLOCATE(ARRAY(N1, N2, N3))
   READ(UNIT) ARRAY
   CLOSE(UNIT)
END SUBROUTINE LOAD
```

仮引き数としてのポインター

仮引き数がポインターの場合、実引き数もポインターになります。また、そのタイプ、型付きパラメーター、およびランクも一致する必要があります。実引き数参照は、ポインター自体に対するもので、ターゲットに対するものではありません。プロシージャーが呼び出されると、次のようになります。
- 仮引き数は、実引き数のポインター関連付け状況を獲得します。
- 実引き数が関連付けられる場合、仮引き数も同じターゲットに関連付けられます。

関連付け状況は、プロシージャーの実行中に変更が可能です。プロシージャーの実行が完了すると、仮引き数の関連付け状況は、関連付けられている場合、未定義になります。

IBM 拡張

引き渡しは参照によって行います。つまり、**%VAL** または **VALUE** をポインター実引き数に指定することはできません。

IBM 拡張 の終り

仮引き数としてのプロシージャー

プロシージャーとして識別される仮引き数を仮プロシージャーと呼びます。仮プロシージャーは、特定の組み込みプロシージャー、モジュール・プロシージャー、外部プロシージャー、または別の仮プロシージャーである実引き数にのみ関連付けられます。組み込みプロシージャーを実引き数として渡す方法の詳細については、 539 ページの『第 12 章 組み込みプロシージャー』を参照してください。

仮プロシージャーおよび対応する実引き数は、両方が関数になるか、または両方がサブルーチンになります。実プロシージャー引き数の仮引き数は、仮プロシージャー引き数の仮引き数と一致していなければなりません。仮引き数が関数である場合、タイプ、型付きパラメーター、ランク、形状 (ポインター以外の配列の場合)、およびポインターであるかどうかが一致しなければなりません。関数の結果の長さが想定される場合、これは結果の特性となります。関数の結果が、定数式でない型付きパラメーターまたは配列の境界を指定する場合、式のエンティティーの依存性は結果の特性です。

サブルーチンである仮プロシージャーは、組み込みデータ型、派生型、および選択戻り指定子とは異なるタイプのように扱われます。このような仮引き数は、サブルーチンまたは仮プロシージャーである実引き数とだけ一致します。

内部サブプログラムを、仮プロシージャー引き数に関連付けることはできません。プロシージャーの参照の規則および制約事項は、 190 ページの『プロシージャー参照』で説明しています。 F95 Fortran 95 では、非組み込みエレメント型プロシージャーを実引き数として使用することはできません。 F95

仮引き数としてのプロシージャーの例

```
PROGRAM MYPROG
INTERFACE
  SUBROUTINE SUB (ARG1)
    EXTERNAL ARG1
    INTEGER ARG1
  END SUBROUTINE SUB
END INTERFACE
EXTERNAL IFUNC, RFUNC
REAL RFUNC

CALL SUB (IFUNC)    ! Valid reference
CALL SUB (RFUNC)    ! Invalid reference
!
! The first reference to SUB is valid because IFUNC becomes an
! implicitly declared integer, which then matches the explicit
! interface. The second reference is invalid because RFUNC is
! explicitly declared real, which does not match the explicit
! interface.
END PROGRAM

SUBROUTINE ROOTS
  EXTERNAL NEG
  X = QUAD(A,B,C,NEG)
```

```
  RETURN
END
FUNCTION QUAD(A,B,C,FUNCT)
  INTEGER FUNCT
  VAL = FUNCT(A,B,C)
  RETURN
END

FUNCTION NEG(A,B,C)
  RETURN
END
```

仮引き数としてのアスタリスク

アスタリスクの形で指定されている仮引き数は、サブルーチン・サブプログラム内の **SUBROUTINE** ステートメントまたは **ENTRY** ステートメントの仮引き数リストにしか指定できません。対応する実引き数は、選択戻り指定子でなければなりません。この指定子は、**CALL** ステートメントと同じ有効範囲にある分岐ターゲット・ステートメントのステートメント・ラベルを示します。

選択戻り指定子の例

```
   CALL SUB(*10)
   STOP                 ! STOP is never executed
10 PRINT *, 'RETURN 1'
   CONTAINS
     SUBROUTINE SUB(*)
          ...
       RETURN 1         ! Control returns to statement with label 10
     END SUBROUTINE
   END
```

プロシージャー参照の解決

プロシージャー参照内のサブプログラム名は、総称名として確立されるか、特定名としてのみ確立されるか、あるいは確立されないかのいずれかです。

以下の条件のうちの 1 つまたは複数が当てはまる場合、サブプログラム名は有効範囲単位内で総称名として確立されます。
- 有効範囲単位に、その名前を持つインターフェース・ブロックがある。
- サブプログラム名が、**INTRINSIC** 属性によって有効範囲単位内で指定された総称組み込みプロシージャーの名前と同じである。
- 有効範囲単位が、使用関連付けを介してモジュールにある総称名を使用している。
- 有効範囲単位内にサブプログラム名の宣言がないが、その名前がホスト有効範囲単位内で総称名として確立されている。

サブプログラム名が総称名として確立されておらず、かつ次のいずれかの条件の 1 つがあてはまる場合、サブプログラム名は有効範囲単位内で特定名としてのみ確立されます。

- 有効範囲単位内に同じ名前を持つインターフェース本体がある。
- 同じ名前を持つステートメント関数、モジュール・プロシージャー、または内部サブプログラムのいずれかが有効範囲単位内にある。
- サブプログラム名が、**INTRINSIC** 属性によって有効範囲単位内で指定された特定組み込みプロシージャーの名前と同じである。
- 有効範囲単位に、サブプログラム名を持つ **EXTERNAL** ステートメントがある。
- 有効範囲単位が、使用関連付けを介してモジュールにある特定名を使用している。
- 有効範囲単位内にサブプログラム名の宣言はないが、その名前がホスト有効範囲単位内で特定名として確立されている。

総称名または特定名のいずれでもない場合、サブプログラム名は確立されません。

名前に対するプロシージャー参照の解決の規則

次の規則を使用して、総称名として確立された名前に対するプロシージャー参照を解決します。

1. 名前のあるインターフェース・ブロックまたは使用関連付けによってアクセス可能なインターフェース・ブロックが有効範囲単位内に存在し、参照がそのインターフェース・ブロックの特定のインターフェースの 1 つへの非エレメント型参照と一致している場合、その参照は特定インターフェースに関連付けられた特定プロシージャーへのものとなります。
2. 1 番目の規則が当てはまらないとき、有効範囲単位内のプロシージャー名が **INTRINSIC** 属性で指定されるか、またはその名前が **INTRINSIC** 属性で指定されるモジュール・エンティティーを使用している場合には、その参照は組み込みプロシージャーに対するものとなります。それで、参照は組み込みプロシージャーのインターフェースと一致します。
3. 規則 1 と 2 の両方ともに当てはまらないが、名前がホストの有効範囲単位内で総称名として確立される場合、その名前はホスト有効範囲単位に規則 1 と 2 を適用することによって解決されます。この規則を適用するには、ホスト有効範囲単位と関数またはサブルーチンのいずれかの名前を持つ有効範囲単位が一致する必要があります。
4. 規則 1、2、3 のいずれにも当てはまらない場合、参照はその名前を持つ総称組み込みプロシージャーに対するものとなります。

次の規則を使用して、特定名として確立された名前に対するプロシージャー参照を解決します。

1. 有効範囲単位がサブプログラムで、その名前を持つインターフェース本体が含まれているか、名前が **EXTERNAL** 属性を持っている場合、および名前がそのプログラムの仮引き数である場合には、その仮引き数は仮プロシージャーとなります。参照はその仮プロシージャーに対するものとなります。
2. 規則 1 が当てはまらず、かつ有効範囲単位にその名前を持つインターフェース本体が含まれているか、名前が **EXTERNAL** 属性を持っている場合、参照は外部サブプログラムに対するものとなります。

3. 有効範囲単位内で、その名前を持つステートメント関数または内部サブプログラムがある場合、参照はそのプロシージャーに対するものとなります。
4. 有効範囲単位で、名前が **INTRINSIC** 属性を持っている場合、参照はその名前を持つ組み込みプロシージャーに対するものとなります。
5. 有効範囲単位には、使用関連付けを介して使用されるモジュール・プロシージャー名に対する参照があります。 **USE** ステートメント内で名前変更の可能性があるため、参照名は元のプロシージャー名とは異なる場合があります。
6. いずれの規則も当てはまらない場合、参照はホスト有効範囲単位にこれらの規則を適用することによって解決されます。

次の規則を使用して、確立されない名前に対するプロシージャー参照を解決します。

1. 有効範囲単位がサブプログラムで、名前がそのサブプログラムの仮引き数名である場合、その仮引き数は仮プロシージャーとなります。参照はその仮プロシージャーに対するものとなります。
2. 規則 1 が当てはまらず、名前が組み込みプロシージャー名の場合、参照はその組み込みプロシージャーに対するものとなります。この規則を適用するには、組み込みプロシージャー定義および関数またはサブルーチンのいずれかの名前を持つ参照が一致していなければなりません。
3. 規則 1 および 2 の両方とも当てはまらない場合、参照はその名前を持つ外部プロシージャーに対するものになります。

総称名に対するプロシージャー参照の解決

総称名へのプロシージャー参照を解決する場合、以下の規則に従います。

- 参照が、同じ名前の総称インターフェース内の特定のインターフェースの 1 つと一貫しており、かつ参照が入っているのと同じ有効範囲単位にあるか、またはその有効範囲単位内にある **USE** ステートメントによってアクセス可能であるかのどちらかである場合、参照はその特定のプロシージャーに対するものになります。
- 最初の規則が適用できず、参照が、同じ名前の総称インターフェース内の特定のインターフェースの 1 つと一貫しており、かつ参照が入っているのと同じ有効範囲単位にあるか、またはその有効範囲単位内にある **USE** ステートメントによってアクセス可能であるかのどちらかである場合、参照はインターフェースを提供するインターフェース・ブロックのその特定のエレメント型プロシージャーに対するものになります。
- 前述の 2 つの規則が当てはまらないとき、有効範囲単位内の名前が **INTRINSIC** 属性で指定されるか、またはその名前が **INTRINSIC** 属性で指定されるモジュールからアクセスできる場合には、その参照は総称プロシージャーに対するものとなります。それで、参照は総称プロシージャーのインターフェースと一致します。
- 前述の 3 つの規則が当てはまらないとき、名前がホストの有効範囲単位内で総称名として確立される場合、その名前はホストの有効範囲単位内に前述の規則を適用する

ことによって解決されます。この規則を適用するには、ホストの有効範囲単位と関数またはサブルーチンのいずれかの名前を持つ有効範囲単位が一致する必要があります。

再帰

直接または間接的に自身を参照することのできるプロシージャーを再帰プロシージャーと呼びます。このようなプロシージャーは、特定の条件を満たすまで、無限にそのプロシージャー自身を参照することができます。たとえば、次のように正の整数 N の階乗を決定することができます。

```
INTEGER N,
RESULT, FACTORIAL
READ (5,*) N
IF (N.GE.0) THEN
  RESULT = FACTORIAL(N)
END IF
CONTAINS
  RECURSIVE FUNCTION FACTORIAL (N) RESULT (RES)
    INTEGER RES
    IF (N.EQ.0) THEN
      RES = 1
    ELSE
      RES = N * FACTORIAL(N-1)
    END IF
  END FUNCTION FACTORIAL
END
```

構文および規則に関する詳細については、 378 ページの『FUNCTION』、 473 ページの『SUBROUTINE』、または 359 ページの『ENTRY』を参照してください。

IBM 拡張

また、コンパイラー・オプション **-qrecur** を指定することによって、外部プロシージャーを再帰的に呼び出すことができます。ただし、XL Fortran は、プロシージャーが **RECURSIVE** または **RESULT** のいずれかのキーワードを指定している場合に、このオプションを無視します。

IBM 拡張 の終り

純粋プロシージャー

Fortran 95

PURE プロシージャーには副次作用がないため、コンパイラーがプロシージャーを呼び出す場合、特定の順序に束縛されません。この例外は次のとおりです。
- 純粋関数。値が返されるからです。

- 純粋サブルーチン。**OUT** または **INOUT** の意図で仮引き数を修正したり、あるいは **POINTER** 属性によって関連付け状況または仮引き数の値を修正できるからです。

純粋プロシージャーは、**FORALL** ステートメントおよび FORALL 構造体で特に有用です。参照されるすべてのプロシージャーに副次作用が起きないように設計されているからです。

次のコンテキストでは、プロシージャーは純粋でなければなりません。
- 純粋プロシージャーの内部プロシージャー
- 定義済み演算子または定義済み割り当てによって参照されたものを含め、 **FORALL** ステートメントまたは **FORALL** 構造体の本体または *scalar_mask_expr* で参照されたプロシージャー
- 純粋プロシージャーで参照されるプロシージャー
- 純粋プロシージャーへのプロシージャー実引き数

組み込み関数 (**RAND**、 XL Fortran 拡張は除く) および **MVBITS** サブルーチンは必ず純粋となります。 **PURE** 属性を持つと明示的に宣言する必要はありません。ステートメント関数は、参照する関数がすべて純粋である場合にのみ、純粋になります。

純粋関数の *specification_part* は、すべての仮引き数が **IN** の意図を持つこと (プロシージャー引き数を除く)、および属性が **POINTER** である引き数を指定しなければなりません。純粋のサブルーチンの *specification_part* は、すべての仮引き数 (プロシージャー引き数を除く)、アスタリスク、および **POINTER** 引き数を持つ引き数の意図を指定しなければなりません。このような純粋プロシージャーのためのインターフェース本体は同様に、仮引き数の意図を指定しなければなりません。

純粋プロシージャーの *execution_part* および *internal_subprogram_part* は、値を変化させるコンテキスト、つまり副次作用を引き起こすコンテキストでは、**IN** の意図を持つ仮引き数、グローバル変数 (または関連したストレージであるオブジェクト)、またはそのサブオブジェクトを参照できません。純粋関数の *execution_part* および *internal_subprogram_part* は、次のようなコンテキストでは、仮引き数、グローバル変数 (または関連したストレージであるオブジェクト)、そのサブオブジェクトを使用してはなりません。
- 変数 がどのレベルでもポインター・コンポーネントを持つ派生型である場合、割り当てステートメントの中の変数、または割り当てステートメントの中の式
- ポインターの割り当てステートメントの中の *pointer_object* または ターゲット
- **DO** または暗黙 **DO** 変数
- **READ** ステートメントの中の *input_item*
- **WRITE** ステートメントの中の内部ファイル識別子
- I/O ステートメントの中の **IOSTAT=** または **SIZE=** 識別変数
- **ALLOCATE**、**DEALLOCATE**、**NULLIFY**、または **ASSIGN** ステートメントの中の変数
- **POINTER** 属性を伴う仮引き数、あるいは **OUT** または **INOUT** の意図と関連した実引き数

- **LOC** への引き数
- **STAT=** 指定子
- **READ** ステートメントで指定する **NAMELIST** の中の変数

純粋プロシージャーは、どのエンティティーも **VOLATILE** であると指定できません。また、**VOLATILE** であるデータへの参照を含めることはできません。含めた場合、使用関連付けまたはホスト関連付けを介してアクセス可能になります。これには、**NAMELIST I/O** を介して生じるデータへの参照も含まれます。

純粋プロシージャーでは、内部 I/O だけを行うことができます。したがって、I/O ステートメントの装置識別子をアスタリスク (*) にしたり、外部装置を参照したりすることはできません。 I/O ステートメントは、**BACKSPACE**、**ENDFILE**、**REWIND**、**OPEN**、**CLOSE**、**INQUIRE**、**READ**、**PRINT**、**WAIT**、および **WRITE** です。 **PAUSE** ステートメントおよび **STOP** ステートメントは、純粋プロシージャーでは使用できません。

純粋関数と純粋サブルーチンには、次の 2 つの違いがあります。

1. サブルーチンのポインター以外の仮データ・オブジェクトは意図を持つ場合もありますが、関数のポインター以外の仮データ・オブジェクトは **IN** の意図を持っていなければなりません。
2. **POINTER** 属性を伴うサブルーチンの仮データ・オブジェクトは、関連付け状況または定義状況、あるいはその両方を変更する場合があります。

プロシージャーが純粋として定義されていない場合、インターフェース本体で純粋と宣言することはできません。しかし、その逆は真ではありません。プロシージャーが純粋と定義される場合、インターフェース本体で純粋と宣言する必要はありません。もちろん、インターフェース本体がプロシージャーを純粋と宣言しない場合、そのプロシージャー (明示的インターフェースを介して参照するとき) は、純粋プロシージャー参照だけが許可されるところでは参照として使用できません (たとえば、**FORALL** ステートメントの中)。

例

```
PROGRAM ADD
  INTEGER ARRAY(20,256)
  INTERFACE                              ! Interface required for
    PURE FUNCTION PLUS_X(ARRAY)          !   a pure procedure
      INTEGER, INTENT(IN) :: ARRAY(:)
      INTEGER :: PLUS_X(SIZE(ARRAY))
    END FUNCTION
  END INTERFACE
  INTEGER :: X
  X = ABS(-4)                            ! Intrinsic function
                                         !   is always pure
  FORALL (I=1:20, I /= 10)
    ARRAY(I,:) = I + PLUS_X(ARRAY(I,:))  ! Procedure references in
                                         !   FORALL must be pure
```

```
  END FORALL
END PROGRAM
PURE FUNCTION PLUS_X(ARRAY)
  INTEGER, INTENT(IN) :: ARRAY(:)
  INTEGER :: PLUS_X(SIZE(ARRAY)),X
  INTERFACE
    PURE SUBROUTINE PLUS_Y(ARRAY)
      INTEGER, INTENT(INOUT) :: ARRAY(:)
    END SUBROUTINE
  END INTERFACE
  X=8
  PLUS_X = ARRAY+X
  CALL PLUS_Y(PLUS_X)
END FUNCTION

PURE SUBROUTINE PLUS_Y(ARRAY)
  INTEGER, INTENT(INOUT) :: ARRAY(:)     ! Intent must be specified
  INTEGER :: Y
  Y=6
  ARRAY = ARRAY+Y
END SUBROUTINE
```

Fortran 95 の終り

エレメント型プロシージャー

Fortran 95

エレメント型サブプログラム定義には、**ELEMENTAL** プレフィックス指定子がなければなりません。 **ELEMENTAL** プレフィックス指定子を使用した場合、**RECURSIVE** 指定子は使用できません。

エレメント型プロシージャーを指定した場合、**-qrecur** オプションは使用できません。

エレメント型サブプログラムは、純粋なサブプログラムです。ただし、純粋なサブプログラムは必ずしもエレメント型サブプログラムではありません。エレメント型サブプログラムの場合、必ずしも **ELEMENTAL** プレフィックス指定子および **PURE** プレフィックス指定子の両方を指定する必要はありません。**PURE** プレフィックス指定子は、**ELEMENTAL** プレフィックス指定子の指定によって暗黙に指定されます。 標準準拠のサブプログラム定義またはインターフェース本体には、**PURE** および **ELEMENTAL** プレフィックス指定子の両方を入れることができます。

エレメント型プロシージャー、サブプログラム、およびユーザー定義エレメント型プロシージャーは、以下の規則に従わなければなりません。

- エレメント型関数の結果はスカラーでなければならず、 **POINTER** または **ALLOCATABLE** 属性を持つことはできません。

- エレメント型サブプログラム内で使用される仮引き数には以下が適用されます。
  - すべての仮引き数はスカラーでなければならず、**POINTER** または **ALLOCATABLE** 属性を持つことはできません。
  - 仮引き数、およびサブオブジェクトは、宣言式内で使用することはできません。ただし、それが **BIT_SIZE**、**KIND**、または **LEN** 組み込み関数への引き数として、または数値照会の組み込み関数として使用される場合を除きます。これについては 539 ページの『第 12 章 組み込みプロシージャー』を参照してください。
  - 仮引き数にアスタリスクを指定することはできません。
  - 仮引き数に仮プロシージャーを指定することはできません。
- エレメント型サブプログラムは、 209 ページの『純粋プロシージャー』で定義されている純粋なサブプログラムに適用されるすべての規則に従わなければなりません。
- エレメント型サブプログラムは、 169 ページの『プログラム単位、プロシージャー、およびサブプログラム』で説明されている純粋なプロシージャーに適用されるすべての規則に従わなければなりません。
- エレメント型サブプログラムには **ENTRY** ステートメントを指定することができますが、 **ENTRY** ステートメントに **ELEMENTAL** プレフィックスを指定することはできません。 **ELEMENTAL** プレフィックスを **SUBROUTINE** または **FUNCTION** ステートメント中に指定すると、 **ENTRY** ステートメントで定義されるプロシージャーはエレメント型になります。
- エレメント型プロシージャーは、要素式の定義済み演算子として使用できますが、 113 ページの『演算子および式』で説明されている要素式の規則に従わなければなりません。

エレメント型プロシージャーへの参照は、以下の場合にのみエレメント型になります。

- 参照がエレメント型関数に対するもので、1 つ以上の実引き数が配列で、すべての配列実引き数が同じ形状を持つ。または
- 参照がエレメント型サブルーチンに対するもので、**INTENT(OUT)** および **INTENT(INOUT)** 仮引き数に対応するすべての実引き数が、同じ形状を持つ配列である。残りの実引き数もこれに従っている。

エレメント型サブプログラムへの参照は、そのすべての引き数がスカラーである場合はエレメント型ではありません。

参照内のエレメント型プロシージャーへの実引き数は、以下のいずれかにすることができます。

- すべてスカラー。エレメント型関数の場合、引き数がすべてスカラーであれば、結果はスカラーになります。
- 1 つ以上の配列値。 1 つ以上の引き数が配列値である場合、以下の規則が適用されます。

- エレメント型関数の場合、結果の形状は最大ランクの配列の実引き数の形状と同じになります。複数の引き数がある場合は、すべての引き数がこれに従っていなければなりません。
- エレメント型サブルーチンの場合、**INTENT(OUT)** および **INTENT(INOUT)** 仮引き数に関連したすべての実引き数は、同じ形状の配列でなければならず、残りの引き数もそれに従わなければなりません。

エレメント型参照の場合、エレメント型の結果の値は、サブルーチンまたは関数が、それぞれの配列の実引き数の対応するエレメントに対して、任意の順序で別々に適用された場合に得られる結果の値と同じになります

組み込みサブルーチン **MVBITS** が使用される場合、**TO** および **FROM** 仮引き数に対応する引き数は、同じ変数にすることができます。これとは別に、エレメント型サブルーチンまたはエレメント型関数への参照内の実引き数は、 196 ページの『引き数関連付け』で説明されている制限事項を満たさなければなりません。

特定のエレメント型プロシージャーがある総称プロシージャーには、特別な規則が適用されます。これについては 208 ページの『総称名に対するプロシージャー参照の解決』を参照してください。

例

例 1:

```
! Example of an elemental function
INTERFACE
  ELEMENTAL REAL FUNCTION LOGN(X,N)
     REAL, INTENT(IN) :: X
     INTEGER, INTENT(IN) :: N
  END FUNCTION LOGN
END INTERFACE

REAL RES(100), VAL(100,100)
  ...
  DO I=1,100
     RES(I) = MAXVAL( LOGN(VAL(I,:),2) )
  END DO
    ...
END PROGRAM P
```

例 2:

```
! Elemental procedure declared with a generic interface
INTERFACE RAND
   ELEMENTAL FUNCTION SCALAR_RAND(x)
     REAL, INTENT(IN) :: X
   END FUNCTION SCALAR_RAND

   FUNCTION VECTOR_RANDOM(x)
     REAL X(:)
     REAL VECTOR_RANDOM(SIZE(x))
   END FUNCTION VECTOR_RANDOM
END INTERFACE RAND

REAL A(10,10), AA(10,10)

! Since the actual argument 'AA' is a two-dimensional array, and
! a specific procedure that takes a two-dimensional array as an
! argument is not declared in the interface block, then the
! elemental procedure 'SCALAR_RAND'  is called.

A = RAND(AA)

! Since the actual argument is a one-dimensional array section, and
! a specific procedure that takes a one-dimensional array as an
! argument is declared in the interface block, then the specific
! one-dimensional procedure 'VECTOR_RANDOM' is called. This is an
! non-elemental reference since 'VECTOR_RANDOM' is not declared to
! be elemental.

A = RAND(AA(6:10,2))

END
```

Fortran 95 の終り

第 8 章 I/O の概念

この章では、以下の項目について説明します。

- 『レコード』
- 218 ページの『ファイル』
- 222 ページの『装置』
- IBM 229 ページの『条件および IOSTAT 値』 IBM

レコード

レコードとは、一連の文字または数値のことです。レコードには、定様式レコード、不定様式レコード、ファイル終了レコードの 3 種類があります。

定様式レコード

定様式レコードは、任意の ASCII 文字ストリングを並べたもので、読み取り可能な形式での印刷が可能です。定様式レコードが読み取られるときは、文字で表現されたデータ値が内部形式に変換されます。定様式レコードが書き込まれるときは、書き込み対象のデータが内部書式から文字に変換されます。

定様式レコードを AIX の **asa** コマンドを使って印刷した場合、レコードの先頭文字によって、垂直スペーシングが決まりますが、この文字は印刷されません。詳細については、「ユーザーズ・ガイド」の『*Fortran ASA キャリッジ制御 (asa) を使用した出力ファイルの印刷*』を参照してください。レコードの残りの文字があれば、左マージンの一番左側の位置から印刷されます。垂直のスペーシングは、リテラル・データの形式で、形式仕様に指定することができます。垂直のスペーシングは、次のように決められています。

レコードの最初の文字	印刷の前の垂直のスペーシング
ブランク	1 行
0	2 行
1	次ページの 1 行目
+	前送りなし

ここに示した文字およびスペーシングは、印刷レコード用に定義されているものです。これ以外の文字をレコードの最初の文字として使用すると、エラーが発生します。印刷レコードに文字が含まれていないと、1 行分のスペースがとられてから、ブランク行が印刷されます。レコードを端末装置に表示する場合にも、レコードの最初の文字が使わ

れますが、所定の行スペースを確保する文字は、ブランク、0、および + だけです。(これらの印刷コードを端末装置に表示させるには、 AIX の **asa** コマンドを使用しなければなりません。詳細については、「ユーザーズ・ガイド」の『*Fortran ASA キャリッジ制御 (asa) を使用した出力ファイルの印刷*』を参照してください。)

不定様式レコード

不定様式レコードは、内部表現形式の値が並んだもので、文字データと文字以外のデータを含むことができるものと、データを含むことのできないものとがあります。値はシステムの内部形式になっていて、読み取り時または書き込み時の値の変換は行われません。

ファイル終了レコード

ファイル終了レコード (存在する場合) は、ファイルの最後のレコードです。このレコードは長さを持ちません。このレコードは、**ENDFILE** ステートメントによって明示的に書き込むことができます。 最終のデータ転送ステートメントが **WRITE** ステートメントであった場合、ファイル終了レコードを、順次アクセスのために接続されたファイルに暗黙的に書き込むことができ、ファイルを参照するファイル位置決めステートメントの介入は行われません。さらに、

- **REWIND** または **BACKSPACE** ステートメントは、ファイルが接続している装置を参照します。あるいは、
- ファイルは、**CLOSE** ステートメントによって明示的にクローズされるか、エラー以外の理由によるプログラムの終了によって暗黙的にクローズされるか、あるいは同じ装置での別の **OPEN** ステートメントによって暗黙的にクローズされます。

ファイル

ファイルとは、一連の内部レコードおよび外部レコードが並んだものです。外部ファイルには、以下の 3 つの方法でアクセスできます。

- 直接アクセス
- 順次アクセス
- ストリーム・アクセス

外部ファイル

外部ファイルとは、ディスク、テープ、または端末装置のような I/O 装置に関連したファイルのことです。

あるプログラムが外部ファイルを読み取り用または書き込み用に使用できるとき、またはプログラムが外部ファイルを作成したとき、実行可能プログラムの外部ファイルが存在するといいます。外部ファイルは、作成すると存在するようになります。外部ファイルは削除されると、存在しなくなります。何も入っていない外部ファイルが存在する可能性もあります。

IBM 拡張

外部ファイルをファイル名で識別する場合、 AIX オペレーティング・システムのファイル名として有効であるためには、各ファイル名の長さを 255 文字以下に、絶対パス名の合計長を 1023 文字以下にする必要があります (ただし、絶対パス名は指定しなくてもかまいません)。

IBM 拡張 の終り

外部ファイルの位置は、通常、先行する I/O 操作によって設定されます。外部ファイルの位置は次のいずれかになります。

- 初期点。先頭のレコードの直前の位置、または最初のファイル記憶単位。
- 終端点。最後のレコードの直後の位置、または最後のファイル記憶単位。
- 現行レコード。レコード内にファイルが位置付けられている場合。それ以外の場合、現行レコードは存在しません。
- 先行レコード。現在のファイル位置の直前のレコード。現行レコードがない場合、前のレコードは現在のファイル位置の直前のレコードです。ファイルが初期点、または最初のレコードに位置付けられているときには、先行レコードはありません。
- 後続レコード。現在のファイル位置の直後のレコード。現行レコードがない場合、次のレコードは現在のファイル位置の直後のレコードです。ファイルが終端点、または最後のレコードに位置付けられているときは、後続レコードはありません。
- 中間点。エラーの後。

外部ファイル・アクセス・モード: 順次、直接、およびストリーム

XL Fortran は、外部ファイルの定数にアクセスする次の 3 つの方法を提供します。

- 順次
- 直接
- ストリーム

ファイルを装置に接続するときのアクセス方式を決定できます。

順次アクセス用に接続されたファイルには、レコードは書き込まれた順に入ります。レコードは、全部が定様式レコードか、全部が不定様式レコードのいずれかでなければなりません。最後のレコードは、ファイル終了レコードでなければなりません。ファイルが順次アクセス用に接続されている間は、直接アクセスまたはストリーム・アクセス I/O ステートメントを使って、レコードを読み取ったり書き込んだりしてはなりません。

直接アクセス用に接続されたファイルのレコードは、任意の順序で読み取りまたは書き込みを行うことができます。レコードは、全部が定様式レコードか、全部が不定様式レコードのいずれかでなければなりません。そのファイルが以前に順次アクセス用に接続されていた場合、ファイルの最後のレコードがファイル終了レコードになる場合があり

ます。この場合、ファイル終了レコードは、そのファイルが直接アクセスまたはストリーム・アクセス用に接続されたときには、ファイルの一部と見なされません。順次またはストリーム・アクセス、リスト指示形式設定、名前リスト形式設定、または非アドバンス I/O ステートメントを使って、レコードの読み取りや書き込みを行うことはできません。

直接アクセス用に接続されているファイルでは、各レコードには、ファイルのレコードの順序を識別するためのレコード番号が付いています。レコード番号は整数値で、レコードの読み取りまたは書き込み時に指定する必要があります。レコードには順に番号が付けられています。最初のレコードが 1 番になります。レコードの読み取りまたは書き込みは、レコード番号順に行う必要はありません。たとえば、9 番、5 番、11 番のレコードを、間にあるレコードを書き込まずに、この順序で書き込むこともできます。

直接アクセス用に接続されたファイル内のレコードは、すべて同じ長さになります。この長さは、ファイルに接続するときに、**OPEN** ステートメント (420 ページの『OPEN』を参照) で指定します。

直接アクセス用に接続されたファイル内のレコードを削除することはできませんが、新しい値に書き直すことはできます。まずレコードを書き込まなければ、レコードを読み取ることはできません。

IBM 拡張

ストリーム・アクセス用のファイルは、定様式としても不定様式としても接続できます。どちらの形式も、1 バイト・ファイル記憶単位で構成される外部ストリーム・ファイルを使用します。不定様式ストリーム・アクセス用に接続されたファイルはストリーム構造のみを持ちますが、定様式ストリーム・アクセス用に接続されたファイルはレコードとストリーム構造の両方を持ちます。これらの二重構造ファイルには以下の特性があります。

- ファイル記憶単位の中にレコード・マーカーを表すものがあります。
- ファイルに保管されているレコード・マーカーからレコード構造が推論されます。
- レコード長に制限がありません。
- レコード・マーカーのない空のレコードの書き込みは無効です。
- ファイル終わりにレコード・マーカーがない場合、最終レコードは空ではなく不完全なものになります。
- 以前に順次アクセス用に接続されたファイル内のファイル終了レコードは、そのファイルをストリーム・アクセス用として接続したときにはファイルの一部とは見なされません。

定様式ストリーム・アクセス用に接続されたファイルの最初のファイル記憶単位の位置が 1 になります。後続の各記憶単位の位置は、直前の記憶単位よりも大きくなります。一連の記憶単位の位置は必ずしも連続するとは限らず、位置指定可能なファイルの読み

取りまたは書き込みを位置順に行う必要はありません。定様式ストリーム・アクセス用に接続されたファイル記憶単位の位置を判別するには、**INQUIRE** ステートメントの **POS=** 指定子を使用します。ファイルを位置指定できない場合は、**INQUIRE** ステートメントを使用して取得した値を使って、そのファイルを位置指定することができます。ファイル作成以降、記憶単位が書き込まれ、接続が **READ** ステートメントを許可する限り、ファイルに接続されている間はファイルからの読み取りを行います。定様式ストリーム・アクセス用に接続されたファイルのファイル記憶単位は、定様式ストリーム・アクセス I/O ステートメントによってのみ、読み取りまたは書き込みを行うことができます。

不定様式ストリーム・アクセス用に接続されたファイルの最初のファイル記憶単位の位置が 1 になります。連続する記憶単位の位置の値は、順番に 1 ずつ増えていきます。位置指定可能なファイルの読み取りまたは書き込みを位置順に行う必要はありません。ファイル作成以降、記憶単位が書き込まれ、接続が **READ** ステートメントを許可する限り、ファイルに接続されている間はファイルから記憶単位を読み取ることができます。不定様式ストリーム・アクセス用に接続されたファイルのファイル記憶単位は、ストリーム・アクセス I/O ステートメントによってのみ、読み取りまたは書き込みを行うことができます。

IBM 拡張 の終り

内部ファイル

内部ファイルは、ベクトル添え字を持つ配列セクションではない文字変数です。内部ファイルは常に存在します。

内部ファイルがスカラー文字変数の場合、ファイルは、そのスカラー変数と等しい長さを持つ 1 つのレコードから成ります。内部ファイルが文字配列の場合、配列の各エレメントは、ファイルのレコードとなり、各レコードの長さは同一になります。

レコードの読み取りおよび書き込みは、順次アクセス定様式 I/O ステートメントによって行われます。内部ファイルを指定できる I/O は、 **READ** および **WRITE** だけです。

WRITE ステートメントによって書き込まれたものが、レコード全体より小さい場合、レコードの残りの部分にブランクが埋め込まれます。

入力において、ブランクは、**BLANK=NULL** を指定してオープンされた外部ファイルの場合と同様に扱われます。レコードには、必要に応じてブランクが埋め込まれます。ただし、これは、**-qxlf77** コンパイラー・オプションの **noblankpad** サブオプションが指定されない場合に限ります。このサブオプションは、レコードに対して埋め込みが行われないことを示します。

内部ファイルのレコードになっているスカラー文字変数は、出力ステートメント以外の方法でも、定義状態または未定義状態にすることができます。たとえば、文字割り当てステートメントを使っても定義状態にすることができます。

装置

装置とは、外部ファイルを参照する手段です。プログラムは、I/O ステートメント内の装置指定子として指定した装置番号によって、外部ファイルを参照します。装置指定子の形式については、446ページを参照してください。

装置の接続

装置と外部ファイルの間の関連付けを接続といいます。接続が行われてからでなければ、ファイルのレコードを読み取ることも、書き込むこともできません。

ファイルと装置を接続するには、次の 3 つの方法があります。
- 事前接続
- 暗黙接続
- **OPEN** ステートメントによる明示接続 (420ページを参照)

事前接続

事前接続は、プログラムの実行が開始されると、一度行われます。事前接続された装置は、事前に **OPEN** ステートメントを実行しなくても、I/O ステートメントで指定することができます。

IBM 拡張

装置 0、5、および 6 が定様式順次アクセス用の名前なしファイルに事前接続されます。
- 装置 0 は標準エラー・デバイスに事前接続されます。
- 装置 5 は標準入力デバイスに事前接続されます。
- 装置 6 は標準出力デバイスに事前接続されます。

これらのファイルのその他の特性は、以下のものを除いて、 **OPEN** 指定子のデフォルト指定子値となります。
- **STATUS='OLD'**
- **ACTION='READWRITE'**
- **FORM='FORMATTED'**

IBM 拡張 の終り

暗黙接続

IBM 拡張

順次の **PRINT**、**READ**、**WRITE**、 **REWIND** または **ENDFILE** ステートメントを、外部ファイルに現在接続されていない装置について実行すると、事前に設定された名前を持つファイルがその装置に暗黙接続されます (デフォルトでは、装置 *n* が **fort.***n* とい

う名前のファイルに接続されます)。装置 0 だけは暗黙的に接続することはできません。これらのファイルは、必ずしも存在している必要はありません。このファイルは、**OPEN** ステートメントを最初に実行しないでそれらの装置を使った場合に作成されます。デフォルトの接続は、順次 I/O 用の接続です。 (異なるファイル名に暗黙接続するための方法については、「ユーザーズ・ガイド」の『*実行時オプションの設定*』にある **UNIT_VARS** 実行時オプションを参照してください。)

装置の外部ファイルへの接続が終了していた場合、事前接続された装置のみが、暗黙的に接続することができます。次の例では、事前接続されていた装置は、暗黙接続が行われる前にクローズされています。

```
      PROGRAM TRYME
      WRITE ( 6, 10 ) "Hello1"   ! "Hello1" written to standard output
      CLOSE ( 6 )
      WRITE ( 6, 10 ) "Hello2"   ! "Hello2" written to fort.6
10    FORMAT (A)
      END
```

暗黙的に接続された装置の特性は、**OPEN** ステートメントのデフォルト指定子の値となります。ただし、この場合、最初のデータ転送ステートメントによって値が決められる **FORM=** および **ASYNCH=** 指定子は除きます。最初の I/O ステートメントで、定様式直接アクセスの形式設定、リスト指示の形式設定、または名前リストの形式設定を使用する場合、 **FORM=** 指定子の値は、**FORMATTED** に設定されます。最初の I/O ステートメントが不定様式の場合は、値は **UNFORMATTED** に設定されます。最初の I/O ステートメントが非同期の場合、**ASYNCH** 指定子の値は **YES** に設定されます。最初の I/O ステートメントが同期の場合は、値は **NO** に設定されます。

IBM 拡張 の終り

切断

CLOSE ステートメントは、装置からファイルを切り離します。 切り離したファイルは、同一のプログラム内で、同じ装置に再接続することも、別の装置に再接続することもできます。

IBM 拡張

装置 0 はクローズできません。装置 5 および 6 は、クローズされた後に、それぞれ、標準入力および標準出力に再接続することはできません。

IBM 拡張 の終り

データ転送ステートメントの実行

READ ステートメントは、外部ファイルまたは内部ファイルのいずれかからデータを取得し、内部記憶装置に格納します。 値は、ファイルから、入力リストに指定されたデータ項目 (指定されていれば) に転送されます。

WRITE ステートメントは、内部記憶装置から取得したデータを、外部ファイルまたは内部ファイルに入れます。 **PRINT** ステートメントは、内部記憶装置から取得したデータを、外部ファイルに入れます。 値は、出力リストおよび形式仕様によって指定されたデータ項目 (指定されていれば) から、ファイルに転送されます。 存在していないファイルに対して **WRITE** または **PRINT** ステートメントを実行すると、エラーが発生していなければ、ファイルが作成されます。

PRINT ステートメント内で出力リストを省略した場合、参照先の **FORMAT** ステートメントの最初の仕様に文字ストリング編集記述子またはスラッシュ編集記述子が含まれていない限り、ブランク・レコードが出力デバイス宛てに転送されます。 この場合、このような仕様によって示されたレコードは、出力デバイスに転送されます。

次に処理すべき項目を決める際には、サイズがゼロの配列および繰り返し回数がゼロの暗黙 **DO** リストは、無視されます。長さがゼロのスカラー文字項目は無視されません。

I/O 項目がポインターの場合、データはファイルと関連したターゲットの間で転送されます。

PAD= 指定子が **NO** の値を持つファイルからのアドバンス入力時に、入力リストおよび形式仕様で、レコード内にある文字を使用することはできません。 **PAD=** 指定子が **YES** の値をもっているか、入力ファイルが内部ファイルの場合に、入力リストおよび形式仕様でレコードにある文字を超える文字を必要とすると、ブランク文字が入れられます。

IBM 拡張

順次アクセス用に接続された外部ファイルに対してのみ埋め込みを行いたい場合は、**-qxlf77** コンパイラー・オプションの **noblankpad** サブオプションを指定します。このサブオプションは、**PAD=** 指定子にはデフォルト値を、直接ファイルおよびストリーム・ファイルには **NO** を、順次アクセス・ファイルには **YES** を指定します。

IBM 拡張 の終り

PAD= 指定子が **NO** の値を持つファイルからの非アドバンス入力時、入力リストおよび形式仕様でレコード内にある文字を超える文字を必要とする場合、レコード終了状態が発生します。 **PAD=** 指定子が **YES** の値を持つ場合に、入力項目および対応するデータ編集記述子がレコード内にある文字を超える文字を必要とすると、レコード終了状態が発生し、ブランク文字が入れられます。レコードが、ストリーム・ファイルの最後のレコードの場合には、ファイルの終わり条件が発生します。

データ転送ステートメントの非同期の実行

IBM 拡張

同期 I/O の場合、その I/O 操作が完了するまで、アプリケーションの実行は停止します。非同期 I/O の場合、アプリケーションは、バックグラウンドの I/O 操作の実行中も処理を続行することができます。

FORTRAN の非同期の **READ** および **WRITE** データ転送ステートメントを使って、非同期データ転送を開始することができます。実際のデータ転送が完了したかどうかに関係なく、非同期 I/O ステートメントの後の実行が続行します。最終的に、データ転送ステートメントの実行の後に、データ転送ステートメント内の **ID=** 変数に戻されたものと同じ **ID=** 値を指定した対応する **WAIT** ステートメントの実行が続かなければなりません。対応する **WAIT** ステートメントの定義については、496 ページの『WAIT』を参照してください。

非同期 I/O ステートメントに指定された I/O 項目の実際のデータ転送は、次に示す時に完了します。
- 非同期データ転送ステートメント時
- 対応する **WAIT** ステートメントの実行の前の任意の時点
- 対応する **WAIT** ステートメント時

ただし、非同期データ転送ステートメントの実行時に実際のデータ転送は完了していなければならない場合があります。このような場合の詳細については、「ユーザーズ・ガイド」の『*XL Fortran I/O のインプリメンテーションの詳細*』を参照してください。

データ転送ステートメントの実行時にエラーが発生した場合、対応する **WAIT** ステートメントは要求されません。**ID=** 値は定義されないからです。この場合、対応する **WAIT** ステートメントの代わりにデータ転送ステートメントが実行されたものとして、エラー処理と状況報告 (**ERR=** および **IOSTAT=**) が行われます。

非同期データ転送ステートメント内に I/O リスト項目として現れる変数や、そのような変数と関連した変数はすべて、対応する **WAIT** ステートメントが実行されるまで、参照したり、定義したり、未定義にしたりしてはなりません。

非同期データ転送ステートメントと、対応する **WAIT** ステートメントとの間での、割り当て可能オブジェクトとポインターの割り振り解除およびポインターの関連付け状況の変更も許可されません。

同一の装置で複数の未解決の非同期データ転送操作があってもかまいませんが、すべて **READ**、またはすべて **WRITE** でなければなりません。同一の装置でのすべての非同期データ転送操作において、対応する **WAIT** ステートメントが実行されるまで、同一の装置での他の I/O 操作はできません。直接アクセスの場合、非同期 **WRITE** ステートメントで、対応する **WAIT** ステートメントが実行されていない非同期 **WRITE** ステートメントと同じ装置とレコード番号を指定してはなりません。ストリーム・アクセスの場

合、非同期 **WRITE** ステートメントで、対応する **WAIT** ステートメントが実行されていない非同期 **WRITE** ステートメントと同じ装置または同じ位置を指定してはなりません。

非同期データ転送ステートメントと、対応する **WAIT** ステートメントとの間で非同期データ転送ステートメントを実行するプログラムの一部では、**NUM=** 指定子の *integer_variable* またはそれに関連するすべての変数を、参照したり、定義したり、未定義にしたりしてはなりません。

例:

```
! A program demonstrating the use of asynchronous I/O
statements.

SUBROUTINE COMPARE(ISTART, IEND, ISIZE, A)
INTEGER, DIMENSION(ISIZE) :: A
INTEGER I, ISTART, IEND, ISIZE
DO I = ISTART, IEND
  IF (A (I) /= I) THEN
    PRINT *, "Expected ", I, ", got ", A(I)
  END IF
END DO
END SUBROUTINE COMPARE

PROGRAM SAMPLE
INTEGER, PARAMETER :: ISIZE = 1000000
INTEGER, PARAMETER :: SECT1 = (ISIZE/2) - 1, SECT2 = ISIZE - 1
INTEGER, DIMENSION(ISIZE), STATIC :: A
INTEGER IDVAR

OPEN(10, STATUS="OLD", ACCESS="DIRECT", ASYNCH="YES", RECL=(ISIZE/2)*4)
A = 0

! Reads in the first part of the array.

READ(10, REC=1) A(1:SECT1)

! Starts asynchronous read of the second part of the array.

READ(10,ID=IDVAR, REC=2) A(SECT1+1:SECT2)

! While the second asynchronous read is being performed,
! do some processing here.

CALL COMPARE(1, SECT1, ISIZE, A)

WAIT(ID=IDVAR)

CALL COMPARE(SECT1+1, SECT2, ISIZE, A)
END
```

関連情報:

- 「ユーザーズ・ガイド」の『*XL Fortran I/O のインプリメンテーションの詳細*』

- 445 ページの『READ』
- 496 ページの『WAIT』
- 500 ページの『WRITE』

IBM 拡張 の終り

アドバンス I/O および非アドバンス I/O

アドバンス I/O は、エラー状態が検出されなければ、最後に読み取りまたは書き込みが行われたレコードの後にレコード・ファイルを位置付けます。

非アドバンス I/O は、現行レコード内の文字位置にファイルを位置付けることができます。非アドバンス I/O により、一連の I/O ステートメントでそれぞれレコードの一部にアクセスすることによって、ファイルの読み取りおよび書き込みを行うことができます。また、可変長レコードを読み取り、そのレコードの長さを照会することもできます。

```
! Reads digits using nonadvancing input

    INTEGER COUNT
    CHARACTER(1) DIGIT
    OPEN (7)
    DO
     READ (7,FMT="(A1)",ADVANCE="NO",EOR=100) DIGIT
       COUNT = COUNT + 1
     IF ((ICHAR(DIGIT).LT.ICHAR('0')).OR.(ICHAR(DIGIT).GT.ICHAR('9'))) THEN
       PRINT *,"Invalid character ", DIGIT, " at record position ",COUNT
       STOP
     END IF
    END DO
100 PRINT *,"Number of digits in record = ", COUNT
    END
```

データ転送が行われる前後のファイルの位置

POSITION= 指定子を指定する順次またはストリーム I/O 用の明示接続 (**OPEN** ステートメントによる) の場合、ファイルは、先頭、最後、または空いている位置に明示的に位置付けられます。

OPEN ステートメントで **POSITION=** 指定子が指定されていない場合は、次のようになります。

- **STATUS=** 指定子が **NEW** または **SCRATCH** の値を持っている場合、ファイルは先頭に位置付けられます。

IBM 拡張

- **STATUS='OLD'** が指定され、**-qposition=appendold** コンパイラー・オプションが指定され、ファイルの位置を変更する次の操作が **WRITE** ステートメントである場合、ファイルは最後に位置付けられます。これらの条件のすべてが満たされない場合、ファイルは先頭に位置付けられます。
- **STATUS='UNKNOWN'** が指定され、 **-qposition=appendunknown** コンパイラー・オプションが指定され、次の操作が **WRITE** ステートメントであれば、ファイルは最後に位置付けられます。これらの条件のすべてが満たされない場合、ファイルは先頭に位置付けられます。

暗黙の **OPEN** の後、ファイルは先頭に位置付けられ、したがって次のようになります。

- ファイルに対する最初の I/O 操作が **READ** である場合、ファイルの最初のレコードが読み取られます。
- ファイルに対する最初の I/O 操作が **WRITE** である場合、ファイルの内容が削除されて、最初のレコードが書き込まれます。

IBM 拡張 の終り

REWIND ステートメントを使用すれば、ファイルを先頭に位置付けることができます。事前接続されている装置 0、5、および 6 は、プログラムの親プロセスから渡されるときに、位置付けられます。

データ転送が行われる前のファイルの位置は、アクセス方式の違いによって次のように異なります。

- 外部ファイルを順次アクセスする場合:
  - アドバンス入力の場合、ファイルは次のレコードの最初に位置付けられます。 このレコードが現行レコードになります。
  - アドバンス出力の場合、新しいレコードが作られ、それがファイルの最後のレコードになります。
- 内部ファイルを順次アクセスする場合: ファイルは最初のレコードの先頭に位置付けられます。 このレコードが現行レコードになります。
- 直接アクセスする場合: ファイルは、レコード指定子で指定されたレコードの始めに位置付けられます。このレコードが現行レコードになります。
- ストリーム・アクセスする場合: ファイルは、**POS=** 指定子で指定されたファイル記憶単位の直前に位置付けられます。**POS=** 指定子がない場合、ファイル位置は変更されません。

アドバンス I/O データ転送が行われた後のファイル位置は、次のとおりです。

- ファイル終了レコードを読み取った結果としてファイルの終わり条件が存在する場合は、終了レコードを超えた位置。

- エラー条件またはファイルの終わり条件が存在しない場合は、最後に読み取りまたは書き込みが行われたレコードを超えた位置。その最後のレコードが先行レコードになります。順次アクセスまたは定様式ストリーム・アクセス用に接続されたファイルに書き込まれたレコードは、ファイルの最後のレコードになります。

非アドバンス入力の場合、エラー条件またはファイルの終わり条件が発生せずに、レコードの終わり条件が発生した場合、ファイルは読み取られたレコードの直後に位置付けられます。非アドバンス入力ステートメントで、エラー条件、ファイルの終わり条件、レコードの終わり条件のいずれも発生しなかった場合、ファイルの位置は変更されません。非アドバンス入力ステートメントで、エラー条件が発生した場合、ファイルの位置は変更されません。これ以外の場合はすべて、ファイルは読み取りまたは書き込みが行われたレコードの直後に位置付けられ、そのレコードが先行レコードになります。

ファイル終了レコードを超えた位置にファイルがある場合、 **READ**、**WRITE**、**PRINT**、または **ENDFILE** ステートメントを実行することはできません (ただし、**-qxlf77=softeof** が設定されていない場合)。 **BACKSPACE** または **REWIND** ステートメントを使用すれば、ファイルの位置を変えることができます。

IBM 拡張

ファイルの終わりを超えて読み取りおよび書き込みができるようにするには、**-qxlf77=softeof** オプションを使用してください。詳細については、「ユーザーズ・ガイド」の **-qxlf77** を参照してください。

IBM 拡張 の終り

エラーのない定様式ストリーム出力については、ステートメントによってデータが転送された先の最も大きな値の位置にファイルの終端点が設定されます。エラーのない不定様式ストリーム出力については、ファイル位置は変更されません。ファイル位置が以前のファイル終端点を超えると、終端点はファイル位置に設定されます。書き込みデータのないファイルの終端点を拡張するには、**POS=** 指定子で空の出力リストを指定してください。データ転送後にエラーが発生した場合は、ファイル位置が不確定になります。

条件および IOSTAT 値

IBM 拡張

IOSTAT 値は、I/O ステートメントの実行時にファイルの終わり条件、レコードの終わり条件、またはエラー条件が発生した場合、 **IOSTAT=** 指定子の変数に割り当てられます。エラー条件には、次に示す 5 つのタイプのものがあります。致命的エラー、重大なエラー、回復可能なエラー、変換、Fortran 90 および Fortran 95 言語。

レコードの終わり条件

I/O ステートメントに **IOSTAT=** 指定子および **EOR=** がある場合、レコードの終わり条件により、**IOSTAT=** 指定子が -4 に設定され、**EOR=** ラベルが分岐します。**IOSTAT=** および **EOR=** 指定子が I/O ステートメントにない場合にレコードの終わり条件が検出されると、プログラムは停止します。

表 6. レコードの終わり条件の IOSTAT 値

IOSTAT 値	レコードの終わり条件の記述
-4	外部ファイルの非アドバンス、形式指示 READ で検出されたレコードの終わり

ファイルの終わり条件

ファイルの終わり条件は、入力リストおよび形式の相互作用により複数のレコードが必要とされるときに、入力ステートメントの実行開始時または定様式または入力ステートメントの実行時に発生します。入力ステートメントに **IOSTAT=** 指定子および **END=** 指定子がある場合、レコードの終わり条件により、**IOSTAT=** 指定子が以下に定義された値のいずれかに設定され、**END=** ラベルが分岐します。 **IOSTAT=** および **END=** 指定子が入力ステートメントにない場合にファイルの終わり条件が検出されると、プログラムは停止します。

ストリーム・アクセスの場合、ファイルの終わりを超えて読み取りを行おうとすると、ファイルの終わり条件が発生します。ファイルの終わり条件は、入力ステートメントの実行開始時、またはストリーム入力ステートメントの実行時に発生します。

ファイルの終わり条件は、定様式アクセス用に接続されたストリーム・ファイルの最終レコードを超えて読み取りを行おうとした場合にも発生します。

表 7. ファイルの終わり条件の IOSTAT 値

IOSTAT 値	ファイルの終わり条件の記述
-1	外部ファイルの順次またはストリーム READ でファイルの終わり、または直接アクセス読み取りで END= が指定されているか、レコードが存在しない。
-2	内部ファイルの READ で検出されたファイルの終わり。

エラー条件

致命的エラー

致命的エラーは、実行システム内で検出されるシステム・レベルのエラーで、このエラーが発生すると、プログラムを実行できなくなります。致命的エラーが発生すると、短い (翻訳されていない) メッセージが装置 0 に書き込まれ、その後に、C ライブラリ

ー・ルーチン **abort()** に対する呼び出しが行われます。メモリー・ダンプの結果は、どのような実行環境が構成されているかによって異なります。

重大なエラー

重大なエラーは、**ERR_RECOVERY** 実行時オプションの値を **YES** に指定していた場合でも、回復させることはできません。 I/O ステートメントに **IOSTAT=** 指定子および **ERR=** がある場合、重大なエラーにより、 **IOSTAT=** 指定子が以下に定義された値のいずれかに設定され、**ERR=** ラベルが分岐します。 I/O ステートメントに **IOSTAT=** および **ERR=** 指定子がない場合に重大エラー条件が検出されると、プログラムは停止します。

表 8. 重大なエラー条件の IOSTAT 値

IOSTAT 値	エラー記述
1	END= が直接アクセス READ に指定されていなくて、レコードが存在しない。
2	内部ファイルの WRITE で検出されたファイルの終わり。
6	ファイルが見つからず、'OLD' が OPEN ステートメントに指定されている。
10	直接ファイルでの読み取りエラー。
11	直接ファイルでの書き込みエラー。
12	順次ファイルまたはストリーム・ファイルでの読み取りエラー。
13	順次ファイルまたはストリーム・ファイルでの書き込みエラー。
14	オープン・ファイル・エラー。
15	ファイルで検出された永続 I/O エラー。
37	動的メモリーの割り振り障害 - メモリー不足。
38	REWIND エラー。
39	ENDFILE エラー。
40	BACKSPACE エラー。
107	ファイルが存在し、STATUS='NEW' が OPEN ステートメントに指定されてた。
119	テープ装置に接続された装置で、BACKSPACE ステートメントを実行しようとした。
122	直接アクセス READ 時に不完全なレコードが検出された。
130	パイプに接続するための OPEN ステートメントに ACTION='READWRITE' が指定された。
135	ユーザー・プログラムが、サポートされないバージョンの XL Fortran 実行時環境に対して呼び出しを行っている。

表 8. 重大なエラー条件の IOSTAT 値 (続き)

IOSTAT 値	エラー記述
139	ACTION= 指定子に正しい値を使ってファイルがオープンされなかったため、I/O 操作が装置上で許可されない。
142	CLOSE エラー。
144	INQUIRE エラー。
152	順次アクセスしかできないファイルに対して、ACCESS='DIRECT' が OPEN ステートメントで指定された。
153	POSITION='REWIND' または POSITION='APPEND' が OPEN ステートメントで指定されたが、ファイルがパイプである。
156	OPEN ステートメントでの RECL= 指定子に対する無効値。
159	関連するデバイスが見つからないため、外部ファイル入力をフラッシュできなかった。
165	次に読み取りまたは書き込みの可能なレコードのレコード番号は、INQUIRE ステートメントの NEXTREC= 指定子で指定された変数の範囲外である。
169	装置は同期 I/O 専用で接続されているため、非同期 I/O ステートメントを完了できない。
172	ファイルは非同期 I/O 不可であるため、接続は失敗した。
173	同じ装置で非同期 WRITE ステートメントの保留中に非同期 READ ステートメントが実行されたか、または同じ装置で非同期 READ ステートメントの保留中に非同期 WRITE ステートメントが実行された。
174	前の非同期 I/O ステートメントが完了していないため、同期 I/O ステートメントを完了できない。
175	ID= 指定子の値が無効であるため、WAIT ステートメントを完了できない。
176	対応する非同期 I/O ステートメントが別の有効範囲単位内にあるため、WAIT ステートメントを完了できない。
178	同じレコードの前の非同期直接 WRITE ステートメントがまだ完了していないため、レコードの非同期の直接 WRITE ステートメントは実行できない。
179	装置上に未完了の非同期 I/O 操作があるため、その装置で I/O 操作はできない。
181	複数接続は同期 I/O でしか許可されないため、ファイルを装置に接続できない。
182	UWIDTH= オプションの値が無効。この値は、32 または 64 でなければならない。

表 8. 重大なエラー条件の IOSTAT 値 (続き)

IOSTAT 値	エラー記述
183	装置の最大レコード長は、INQUIRE ステートメントの RECL= 指定子で指定されたスカラー変数の範囲外である。
184	送信データのバイト数は、I/O ステートメントの SIZE= または NUM= 指定子で指定されたスカラー変数の範囲外である。
185	ファイルを、それぞれ異なる UWIDTH 値を持つ 2 つの装置に接続できない。
186	装置番号は、0 ～ 2,147,483,647 の間になければばならない。
192	ファイル位置の値が、INQUIRE ステートメントの POS= 指定子で指定されたスカラー変数の範囲外である。
193	ファイル・サイズの値が、INQUIRE ステートメントの SIZE= 指定子で指定されたスカラー変数の範囲外である。

回復可能エラー

回復可能エラーは、回復できるエラー状態のことです。 I/O ステートメントに **IOSTAT=** 指定子および **ERR=** がある場合、回復可能エラーが発生すると、 **IOSTAT=** 指定子が以下に定義された値のいずれかに設定され、**ERR=** ラベルへ分岐します。 I/O ステートメントに **IOSTAT=** および **ERR=** 指定子がなく、**ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、回復処理が行われ、プログラムは継続します。 I/O ステートメントに **IOSTAT=** および **ERR=** 指定子がなく、**ERR_RECOVERY** オプションが **NO** に設定されていると、プログラムは停止します。

表 9. 回復可能エラー条件の IOSTAT 値

IOSTAT 値	エラー記述
16	直接 I/O で REC= 指定子の値が無効。
17	直接ファイルで I/O ステートメントが許可されない。
18	未接続の装置での直接 I/O ステートメント。
19	定様式ファイルで不定様式 I/O を行おうとした。
20	不定様式ファイルで定様式ファイル I/O を行おうとした。
21	直接ファイルで順次 I/O またはストリーム I/O を行おうとした。
22	順次ファイルまたはストリーム・ファイルで直接 I/O を行おうとした。
23	すでに別の装置に接続済みのファイルに接続しようとした。
24	OPEN 指定子が接続されたファイルの属性と一致しない。
25	直接ファイルについて、OPEN ステートメントで RECL= 指定子が省略された。
26	OPEN ステートメントの RECL= 指定子が負の値をとる。

表 9. 回復可能エラー条件の IOSTAT 値 (続き)

IOSTAT 値	エラー記述
27	OPEN ステートメントの ACCESS= 指定子が無効である。
28	OPEN ステートメントの FORM= 指定子が無効である。
29	OPEN ステートメントの STATUS= 指定子が無効である。
30	OPEN ステートメントの BLANK 指定子が無効である。
31	OPEN または INQUIRE ステートメントの FILE= 指定子が無効である。
32	STATUS='SCRATCH' および FILE= 指定子が同一の OPEN ステートメントに指定された。
33	ファイルが STATUS='SCRATCH' によってオープンされたときに、STATUS='KEEP' が CLOSE ステートメントに指定された。
34	CLOSE ステートメントの STATUS= 指定子の値が無効である。
36	I/O ステートメントで誤った装置番号が指定された。
47	名前リスト入力項目がゼロ以外のランクの 1 つ以上のコンポーネントで指定された。
48	名前リスト入力項目がゼロ・サイズの配列で指定された。
58	様式指定のエラー。
93	エラー装置 (装置 0) で I/O ステートメントが許可されなかった。
110	定様式 I/O のデータ項目で、無効な編集記述子が使用された。
120	NLWIDTH の設定値がレコードの長さを超えた。
125	不定様式ファイルの OPEN ステートメントで、BLANK= 指定子が指定された。
127	直接ファイルの OPEN ステートメントで、POSITION= 指定子が指定された。
128	OPEN ステートメントの POSITION= 指定子の値が無効である。
129	OPEN ステートメントの ACTION= 指定子が無効である。
131	不定様式ファイルの OPEN ステートメントで、DELIM== 指定子が指定された。
132	OPEN ステートメントの DELIM= 指定子の値が無効である。
133	不定様式ファイルの OPEN ステートメントで、PAD= 指定子が指定された。
134	OPEN ステートメントの PAD= 指定子の値が無効である。
136	READ ステートメントの ADVANCE= 指定子の値が無効である。
137	SIZE= が READ ステートメントに指定されるときに、ADVANCE='NO' が指定されていない。

表 9. 回復可能エラー条件の IOSTAT 値 (続き)

IOSTAT 値	エラー記述
138	EOR= が READ ステートメントに指定されるときに、ADVANCE='NO' が指定されていない。
145	ファイルがファイル終了レコードの後に位置付けられているときに、READ または WRITE を実行しようとした。
163	非ランダム・アクセス装置上にあるファイルへの複数の接続は行えない。
164	ACTION='WRITE' または ACTION='READWRITE' を指定した複数の接続は行えない。
170	OPEN ステートメントの ASYNCH= 指定子の値が無効である。
171	FORM= 指定子が FORMATTED に設定されているため、OPEN ステートメントに指定された ASYNCH= 指定子は無効である。
177	未完了の非同期 I/O 操作があるときに、装置がクローズされた。
191	ACCESS= 指定子に STREAM 値を持つ OPEN ステートメントで RECL= 指定が指定された。
194	BACKSPACE ステートメントが不定様式ストリーム I/O 用に接続された装置を指定した。
195	I/O ステートメントの POS= 指定子が 1 よりも小さい。
196	ストリーム・アクセス用に装置が接続されていないため、その装置上でストリーム I/O ステートメントを実行できない。
197	I/O ステートメントの POS= 指定子が、検出できないファイルに接続されている装置を指定した。
198	未接続の装置でのストリーム I/O ステートメント。

変換エラー

データが無効であるか、データ転送ステートメントでデータの長さが正しくないと、変換エラーが発生します。 I/O ステートメントに **IOSTAT=** 指定子および **ERR=**ラベルがあり、 **CNVERR** オプションが **YES** に設定されている場合、変換エラーが発生すると、**IOSTAT=** 指定子が以下に定義した値のいずれかに設定され、**ERR=** ラベルに分岐します。 I/O ステートメントに **IOSTAT=** および **ERR=** 指定子がなく、**CNVERR** および **ERR_RECOVERY** の両オプションが **YES** に設定されていると、回復処理が行われ、プログラムは継続します。 I/O ステートメントに **IOSTAT=** および **ERR=** 指定子がなく、**CNVERR** オプションが **YES** に設定され、**ERR_RECOVERY** オプションが **NO** に設定されていると、プログラムは停止します。 **CNVERR** オプションが **NO** に設定されると、**ERR** ラベルには分岐せずに、以下に示すように **IOSTAT=** 指定子が設定される場合があります。

表 10. 変換エラー条件の IOSTAT 値

IOSTAT 値	エラー記述	CNVERR=NO の場合の IOSTAT の設定
3	不定様式ファイルでレコードの終わりが検出された。	なし
4	アドバンス I/O を使用する定様式ファイルでレコードの終わりが検出された。	なし
5	内部ファイルでレコードの終わりが検出された。	なし
7	外部ファイルで誤った形式のリスト指示入力が検出された。	あり
8	内部ファイルで誤った形式のリスト指示入力が検出された。	あり
9	内部ファイルに対してリスト指示または NAMELIST データ項目が長すぎる。	あり
41	外部ファイルで有効な論理入力が検出されなかった。	なし
42	内部ファイルで有効な論理入力が検出されなかった。	なし
43	外部ファイルでリスト指示または NAMELIST 入力の使用が予想される複素数値が検出されなかった。	なし
44	内部ファイルでリスト指示または NAMELIST 入力の使用が予想される複素数値が検出されなかった。	なし
45	NAMELIST 入力で、不明または誤った派生型のコンポーネント名で NAMELIST 項目名が指定された。	なし
46	NAMELIST 入力で、誤ったサブストリング範囲で NAMELIST 項目名が指定された。	なし
49	リスト指示または名前リスト入力で誤って区切られた文字ストリングが入っていた。	なし
56	誤った数字が B、O、または Z 形式の編集記述子の入力で検出された。	なし
84	外部ファイルで NAMELIST グループ・ヘッダーが検出されなかった。	あり
85	内部ファイルで NAMELIST グループ・ヘッダーが検出されなかった。	あり
86	外部ファイルで誤った NAMELIST 入力値が検出された。	なし
87	内部ファイルで誤った NAMELIST 入力値が検出された。	なし

表 10. 変換エラー条件の IOSTAT 値 (続き)

IOSTAT 値	エラー記述	CNVERR=NO の場合の IOSTAT の設定
88	NAMELIST 入力で誤った名前が検出された。	なし
90	入力で、NAMELIST グループまたは項目名に誤った文字がある。	なし
91	NAMELIST の入力構文が無効である。	なし
92	入力で、NAMELIST 項目に対して添え字リストが無効である。	なし
94	外部ファイルで、リスト指示または NAMELIST 入力に対して繰り返し指定子が無効である。	なし
95	内部ファイルで、リスト指示または NAMELIST 入力に対して繰り返し指定子が無効である。	なし
96	入力での整数のオーバーフロー。	なし
97	入力で 10 進数字が無効である。	なし
98	B、O、または Z 形式の編集記述子に対して入力データが長すぎる。	なし
121	NAMELIST 項目名または NAMELIST グループ名の出力長が、最大レコード長または NLWIDTH オプションで指定した出力幅よりも長い。	あり

Fortran 90 および Fortran 95 言語エラー

Fortran 90 言語のエラーは、Fortran 90 言語に対する XL Fortran の拡張機能を使用することによって起こります。これは、コンパイル時には検出できません。 **LANGLVL** 実行時オプションが **90STD** という値で指定されていて、 **ERR_RECOVERY** 実行時オプションが設定されていないか、 **NO** に設定されている場合に、Fortran 90 言語エラーは、重大なエラーと見なされます。 **LANGLVL=90STD** および **ERR_RECOVERY=YES** が両方とも指定されている場合には、エラーは回復可能エラーと見なされます。 **LANGLVL= EXTENDED** が指定されると、そのエラー条件はエラーとは見なされません。

Fortran 95 言語のエラーは、Fortran 95 言語に対して、 XL Fortran の拡張機能を使用することによって起こります。これは、コンパイル時には検出できません。 **LANGLVL** 実行時オプションが **95STD** という値で指定されていて、**ERR_RECOVERY** 実行時オプションが設定されていないか、**NO** に設定されている場合に、Fortran 95 言語エラーは、重大なエラーと見なされます。**LANGLVL=95STD** および **ERR_RECOVERY=YES** が両方とも指定されている場合には、エラーは回復可能エラーと見なされます。

LANGLVL= EXTENDED が指定されると、そのエラー条件はエラーとは見なされません。

表 11. Fortran 90 および Fortran 95 言語エラー条件の IOSTAT 値

IOSTAT 値	エラー記述
53	定様式 I/O での編集記述子と項目タイプの不一致。
58	様式指定のエラー。
140	I/O ステートメントを実行しようとしたときに、装置が接続されていない。これは、READ、WRITE、PRINT、REWIND、および ENDFILE の場合のみ。
141	装置の REWIND または BACKSPACE の介入なしに 2 つの ENDFILE ステートメントがある。
151	FILE= 指定子が抜けており、OPEN ステートメントで STATUS= 指定子が 'SCRATCH' の値をもっていない。
187	NAMELIST 注釈は Fortran 90 標準では使用できません。
199	STREAM が、Fortran 90 または Fortran 95 での OPEN ステートメントの ACCESS= に有効な値ではない。

IBM 拡張 の終り

第 9 章 I/O の形式設定

定様式の **READ**、**WRITE**、および **PRINT** ステートメントは、形式設定についての情報を使って、内部データ表現と定様式レコード内の文字表現との間の編集処理 (変換) を指示します (372 ページの『FORMAT』を参照してください)。

この章では、以下の項目について説明します。

- 『形式指示の形式設定』
- 243 ページの『編集』
- 269 ページの『I/O リストと形式仕様の相互作用』
- 270 ページの『リスト指示の形式設定』
- 274 ページの『名前リストの形式設定』

形式指示の形式設定

形式指示の形式設定では、形式仕様内の編集記述子によって編集処理が制御されます。形式仕様は、 **FORMAT** ステートメントの中で指定するか、あるいはデータ転送ステートメント内の文字配列の値として指定します。

データ編集記述子

形式	使途	ページ
A **A***w*	文字値を編集します。	244
B*w* **B***w.m*	2 進値を編集します。	245
E*w.d* **E***w.d***E***e* **E***w.d***D***e* * **E***w.d***Q***e* * **D***w.d* **EN***w.d* **EN***w.d***E***e* **ES***w.d* **ES***w.d***E***e* **Q***w.d* *	指数付き実数および複素数を編集します。	247
F*w.d*	指数なしの実数および複素数を編集します。	251

形式	使途	ページ
G*w.d* **G***w.d***E***e* **G***w.d***D***e* * **G***w.d***Q***e* *	データのタイプに出力形式を適用し、組み込みタイプのデータ・フィールドを編集します。また、データのタイプが実数の場合、データの絶対値を編集します。	253
I*w* **I***w.m*	整数を編集します。	255
L*w*	論理値を編集します。	256
O*w* **O***w.m*	8 進値を編集します。	257
Q *	入力レコード内に残っている文字のカウントを戻します。 *	259
Z*w* **Z***w.m*	16 進値を編集します。	260

注: * IBM 拡張

それぞれの意味は次のとおりです。

w すべてのブランクを含む、フィールドの幅を指定します。これは正でなければなりません。 F95 ただし、出力で、**I**、**B**、**O**、**Z**、および **F** 編集記述子でこれにゼロを指定できる、 Fortran 95 は例外です。 F95

m 印刷する桁数を示します。

d 小数点以下の桁数を指定します。

e 指数フィールド内の桁数を指定します。

w、*m*、*d*、*e* は、以下のように表すこともできます。
- 無符号のリテラル整数

IBM 拡張

- 不等号括弧 (**<** と **>**) で囲まれているスカラー整数式。 詳細については、 377 ページの『変数形式設定式』を参照してください。

IBM 拡張 の終り

w、*m*、*d*、または *e* に対して kind パラメーターを指定することはできません。

IBM 拡張

注:

Q データ編集記述子には 2 つのタイプがあります (**Q***w.d* および **Q**)。

拡張精度 Q
: **Q***w.d* という構文からなる **Q** 編集記述子です。

文字カウント Q
: **Q** という構文からなる **Q** 編集記述子です。

IBM 拡張 の終り

制御編集記述子

形式	使途	ページ
/ *r* /	現在のレコードに関するデータ転送の終わりを指定します。	262
:	I/O リスト内にこれ以上項目がない場合に、形式制御の終わりを指定します。	262
$ *	出力でレコードの終わりを抑制します。*	263 *
BN	数値入力フィールド内の非先行ブランクを無視します。	264
BZ	数値入力フィールド内の非先行ブランクをゼロとして解釈します。	264
*k***P**	実数および複素数項目に対してスケール因数を指定します。	266
S SS	正符号を書き込まないように指定します。	267
SP	正符号を書き込むように指定します。	267
T*c*	次の文字の転送先または転送元のレコード内での絶対位置を指定します。	268
TL*c*	次の文字の転送先または転送元のレコード内での相対位置 (レコード内の現在の位置の左側の位置) を指定します。	268
TR*c*	次の文字の転送先または転送元のレコード内での相対位置 (レコード内の現在の位置の右側の位置) を指定します。	268

形式	使途	ページ
*o***X**	次の文字の転送先または転送元のレコード内での相対位置 (レコード内の現在の位置の右側の位置) を指定します。	268

注: * IBM 拡張

それぞれの意味は次のとおりです。

r　　繰り返し指定子です。これは、無符号かつ正のリテラル整定数です。

k　　使用するスケール因数を指定します。これは、オプションの符号付きリテラル整定数です。

c　　レコード内の文字位置を指定します。これは、無符号かつゼロ以外のリテラル整定数です。

o　　レコード内の相対文字位置です。これは、無符号かつゼロ以外のリテラル整定数です。

IBM 拡張

r、*k*、*c*、および *o* は、整数値に計算される不等号括弧 (**<** と **>**) で囲まれた算術式としても表すことができます。

IBM 拡張 の終り

r、*k*、*c*、*o* に対して kind 型付きパラメーターを指定することはできません。

文字ストリング編集記述子

形式	使途	ページ
*n***H***str*	文字ストリング (str) を出力します。	265
'*str*' "*str*"	文字ストリング (*str*) を出力します。	263

n　　リテラル・フィールド内の文字数です。これは、無符号かつ正のリテラル整定数です。ブランクは文字のカウントに含まれます。 kind 型付きパラメーターを指定することはできません。

編集

編集はフィールド上で行われます。フィールドは、形式制御がデータまたは文字ストリング編集記述子を処理する際に入力で読み取られ、出力で書き込まれるレコードの一部です。フィールドの幅は文字のフィールドのサイズです。

I、**F**、**E**、**EN**、 **ES**、**B**、**O**、**Z**、**D**、**G**、および拡張精度 **Q** 編集記述子は、まとめて数値編集記述子と呼ばれます。これらの編集記述子を使って、整数、実数および複素数のデータの形式を設定します。これらの編集記述子に適用される一般的な規則として、次のものがあります。

- 入力の場合:
  - 先行ブランクは無視されます。その他のブランクの解釈は、 **OPEN** ステートメントの中の **BLANK=** 指定子、および **BN** および **BZ** の 2 つの編集記述子によって制御されます。すべてのブランクのフィールドは、ゼロと見なされます。プラス記号の指定はオプションですが、**B**、**O**、および **Z** 編集記述子に対して指定することはできません。
  - **F**、**E**、**EN**、**ES**、**D**、**G**、および拡張精度 **Q** 編集では、入力フィールド内にある小数点は、小数点位置を指定する編集記述子の部分をオーバーライドします。フィールドでは、内部的に表現できる桁数を超える桁を持つことができます。
- 出力の場合:
  - 文字は、フィールド内では右寄せに入れられます。編集処理による文字数がフィールド幅を下回る場合、フィールドに対して先行ブランクが入れられます。文字数がフィールド幅を超える場合、または指数が指定の長さを超える場合、フィールド全体がアスタリスクで埋められます。
  - 負の値の前には、負符号が付けられます。デフォルトの場合、正またはゼロの値には、符号は付きません。 **S**、**SP**、および **SS** 編集記述子によって制御されると、正符号が値の前に付けられます。

Fortran 95

- **-qxlf90** コンパイラー・オプションの **signedzero** サブオプションと **nosignedzero** サブオプションのどちらを指定するかに応じて、**E**、**D**、**Q (拡張精度)**、**F**、**EN**、**ES**、または **G (一般編集)** 編集記述子の結果は以下のようになります。
  - **signedzero** サブオプションを選択した場合、出力に関する内部値が負の数か負のゼロであれば、出力フィールドに負符号が必ず書き出されます。これは、出力値がゼロであっても変わりません。 Fortran 95 標準ではこの処理は必須です。

    IBM 拡張

    XL Fortran では、**REAL(16)** 内部値がゼロの場合にゼロが負のゼロとして扱わ

れることは決してありません。

IBM 拡張 の終り

- **nosignedzero** サブオプションを選択した場合、出力値がゼロであれば、出力フィールドに負符号が書き出されません。これは、内部値が負の数であっても変わりません。 Fortran 90 標準ではこの処理は必須で、XL Fortran の処理と一致します。

Fortran 95 の終り

IBM 拡張

– XL Fortran では、 `NaN` (番号ではない) が `'NaNQ'`、 `'+NaNQ'`、 `'-NaNQ'`、 `'NaNS'`、 `'+NaNS'`、または `'-NaNS'` によって示されます。無限大は、`'INF'`、 `'+INF'`、または `'-INF'` によって示されます。

IBM 拡張 の終り

注:

1. 内部値がゼロでない場合、 **signedzero** と **nosignedzero** のどちらの場合でも **ES** および **EN** 編集記述子は同じ処理を行います。 したがって、値が負の数である場合は必ず負符号が印刷出力されます。
2. 編集記述子の例において、出力 桁内の小文字 `b` はその位置にブランクが表示されることを示します。

複素数編集

複素数は、1 対の別個の実数コンポーネントからなる値です。したがって、複素数の編集も 1 対の編集記述子によって指定されます。最初の編集記述子は、数値の実数部を編集し、2 番目の編集記述子は虚数部を編集します。 2 つの編集記述子は同じでも、異なっていてもかまいません。 2 つの編集記述子の間に、1 つまたは複数の制御編集記述子を置くことが可能ですが、データ編集記述子を指定することはできません。

データ編集記述子

A (文字) 編集

形式:

A
A*w*

A 編集記述子は、文字値の編集を指示します。これは、文字タイプまたは他の任意のタイプの I/O リスト項目に対応します。転送および変換されるすべての文字の kind 型付きパラメーターは、対応するリスト項目によって暗黙的に示されます。

入力の場合、*w* が入力リスト項目の長さ (*len* と呼びます) 以上であれば、入力フィールドの右端の *len* 個の文字が取られます。指定したフィールド幅が *len* よりも小さければ、*w* 個の文字が左寄せされ、(*len* - *w*) の後続ブランクが追加されます。

出力の場合、*w* が *len* より大きい場合、出力フィールドは、(*w* - *len*) とそれに続く、内部表現からの *len* 個の文字で構成されます。 *w* が *len* 以下であれば、出力フィールドは、内部表現からの左端の *w* 個の文字で構成されます。

w を指定しないと、文字フィールドの幅は、対応する I/O リスト項目の長さになります。

定様式ストリームへのアクセス時、文字出力に改行文字が含まれている場合、その文字出力は複数のレコードに分かれる可能性があります。

B (2 進) 編集

形式:

B*w*
B*w***.***m*

B 編集記述子は、内部形式の任意のタイプの値とその値を 2 進で表現したものとの間の編集を指示します。 (2 進数字は、0 または 1 です。)

入力の場合、*w* 個の 2 進数字が編集され、入力リスト項目の値の内部表現が作られます。入力フィールド内の 2 進数字は、入力リスト項目に割り当てられた値の内部表現の右寄り部分の 2 進数字に対応します。入力の場合、*m* は機能しません。

出力の場合、*w* はゼロよりも大きくなければなりません。

Fortran 95

出力の場合、*w* はゼロでもかまいません。 *w* がゼロの場合、出力フィールドは出力値を表すために必要な文字の最小数で構成されます。

Fortran 95 の終り

B*w* の出力フィールドは、ゼロ個以上の先行ブランクに続いて内部値を先行ゼロなしの 2 進数字の形式で表記したもので構成されます。 2 進定数は、常に 1 桁以上から構成されることに注意してください。

B*w.m* の出力フィールドは、数字ストリングが *m* 桁以上であるという点を除いて、**B***w* と同じになります。必要に応じて、数字ストリングには、先行ゼロが埋められます。 *m* の値は、*w* の値がゼロでない限り、*w* の値を超えてはなりません。 *m* がゼロで、内部データの値がゼロの場合、出力フィールドは、符号の制御が有効であるか否かにかかわらず、ブランク文字のみで構成されます。

m がゼロ、*w* が正の数で、内部データの値がゼロの場合、出力フィールドは、*w* 個のブランク文字で構成されます。 F95 *w* と *m* の両方がゼロで、内部データの値がゼロの場合、出力フィールドは 1 つのブランク文字だけで構成されます。 F95

-qxlf77 コンパイラー・オプションの **nooldboz** サブオプションが指定される場合 (デフォルト)、出力フィールド幅が出力全体を包含するのに不十分であるとき、アスタリスクが印刷されます。入力の場合、**BN** および **BZ** 編集記述子は、 **B** 編集記述子に影響します。

IBM 拡張

-qxlf77 コンパイラー・オプションの **oldboz** サブオプションを指定すると、出力で以下の処理が行われます。

- *m* を *w* の最小値およびデータ項目の最大可能値を表現するために必要な桁数の値と想定すると、 **B***w* は **B***w.m* として扱われます。
- 出力はブランクとそれに続く *m* 桁以上のデータから構成されます。必要に応じて、*m* 桁の数字になるまで右端からゼロが埋められます。数字が大きすぎて出力フィールドに入らないと、右端の *m* 桁分の数字が出力されます。

w がゼロの場合、**oldboz** サブオプションは無視されます。

oldboz サブオプションによって、**BN** および **BZ** 編集記述子は、**B** 編集記述子に影響しません。

IBM 拡張 の終り

入力の場合の B 編集の例

```
Input      Format      Value
111        B3          7
110        B3          6
```

出力の場合の B 編集の例

```
Value      Format          Output                  Output
                   (-qxlf77=oldboz の指定)  (-qxlf77=nooldboz の指定)
 7          B3             111                     111
 6          B5             00110                   bb110
17          B6.5           b10001                  b10001
17          B4.2           0001                    ****
```

```
22          B6.5          b10110                b10110
22          B4.2          0110                  ****
 0          B5.0          bbbbb                 bbbbb
 2          B0            10                    10
```

E、D、および Q (拡張精度) 編集

形式:

E*w***.***d*

E*w***.***d***E***e*

D*w***.***d*

IBM **E***w***.***d* **D***e* IBM

IBM **E***w***.***d* **Q***e* IBM

IBM **Q***w***.***d* IBM

E、**D**、および拡張精度 **Q** 編集記述子は、内部形式の実数および複素数と、それを指数付きの文字表現にしたものとの間の編集を指示します。 **E**、**D**、または拡張精度 **Q** 編集記述子は、実数タイプの I/O リスト項目、複素数タイプの I/O リスト項目の実数部または虚数部、 IBM または、長さが 4 バイト以上の XL Fortran のそれ以外のタイプに対応します。 IBM

入力フィールドの形式は、 **F** 編集の場合と同じです。入力の場合、*e* は機能しません。

スケール因数の出力フィールドの形式は、次のとおりです。

```
►►─┬─────┬──┬───┬─.─digit_string─decimal_exponent─────────────────►◄
   ├─ + ─┤  └ 0 ┘
   └─ - ─┘
```

digit_string
丸められた後で、*d* 桁の最初の有効数字になる数字ストリングです。

decimal_exponent
次のいずれかの形式を持つ 10 進表示の指数です (*z* は 10 進数字です)。

編集記述子	指数の絶対値 (スケール因数が 0)	指数の形式
E*w.d*	|decimal_exponent| ≤ 99	$\mathbf{E}\pm z_1z_2$
E*w.d*	99<|decimal_exponent| ≤ 309	$\pm z_1z_2z_3$
E*w.d***E***e*	|decimal_exponent| ≤ (10^e)-1	$\mathbf{E}\pm z_1z_2\cdots z_e$
E*w.d***D***e* *	|decimal_exponent| ≤ (10^e)-1 *	$\mathbf{D}\pm z_1z_2\cdots z_e$ *
E*w.d***Q***e* *	|decimal_exponent| ≤ (10^e)-1 *	$\mathbf{Q}\pm z_1z_2\cdots z_e$ *
D*w.d*	|decimal_exponent| ≤ 99	$\mathbf{D}\pm z_1z_2$

編集記述子	指数の絶対値 (スケール因数が 0)	指数の形式
D_w.d_	99<|decimal_exponent| ≤ 309	$\pm z_1z_2z_3$
Q_w.d_ *	|decimal_exponent| ≤ 99 *	**Q**$\pm z_1z_2$ *
Q_w.d_ *	99<|decimal_exponent| ≤ 309 *	$\pm z_1z_2z_3$ *

注: * IBM 拡張

スケール因数 *k* (266 ページの『P (スケール因数) 編集』を参照) は、10 進数の正規化を制御します。 $-d<k\leq0$ ならば、出力フィールドには $|k|$ 個の先行ゼロが入り、小数点以下の有効数字の桁数は $d - |k|$ 桁です。 $0<k<d+2$ ならば、出力フィールドには、小数点の左側に *k* 桁の有効数字が入り、小数点の右側に $d-k+1$ が入ります。これ以外の *k* の値は使用できません。

詳細については、243ページの数値編集に関する一般情報を参照してください。

IBM 拡張

注: 実数編集記述子を使って表示する値が、表示可能な数の範囲外にある場合、XL Fortran では、次のものを表示する ANSI/IEEE 浮動小数点形式がサポートされています。

表 12. 浮動小数点表示

表示	意味
NaNQ +NaNQ	正の静止 NaN (番号でない)
-NaNQ	負の静止 NaN
NaNS +NaNS	正のシグナル NaN
-NaNS	負のシグナル NaN
INF +INF	正の無限大
-INF	負の無限大

IBM 拡張 の終り

入力における E、D、および拡張精度 Q 編集の例

(ブランクの解釈には **BN** 編集が有効であると仮定します。)

```
Input      Format     Value
12.34      E8.4       12.34
·1234E2    E8.4       12.34
2.E10      E12.6E1    2.E10
```

出力における E、D、および拡張精度 Q 編集の例

```
Value      Format           Output                       Output
                     (-qxlf77=noleadzero の指定)  (-qxlf77=leadzero の指定)
1234.56    E10.3        bb.123E+04                   b0.123E+04
1234.56    D10.3        bb.123D+04                   b0.123D+04
```

Fortran 95

```
                  (-qxlf90=signedzero の指定) (-qxlf90=nosignedzero の指定)
-0.001     E5.2         -0.00                        b0.00
```

Fortran 95 の終り

EN 編集

形式:

EN*w.d*
EN*w.d***E***e*

EN 編集記述子は、出力値がゼロの場合を除いて、10 進指数が 3 で割り切れ、有効数字部の絶対値が 1 以上、1000 未満である技術表記による実数値形式で出力フィールドを作り出します。スケール因数は、出力には影響しません。

EN 編集記述子は、実数タイプの I/O リスト項目、複素数タイプの I/O リスト項目の実数部または虚数部、 IBM または長さが 4 バイト以上の XL Fortran のそれ以外のタイプに対応します。 IBM

入力フィールドの形式および解釈は、F 編集の場合と同じです。

出力フィールドの形式は、次のとおりです。

```
►►──┬─────┬──┬─────┬──.──digit_string──exp──────────────────────►◄
    ├─ + ─┤  └ yyy ┘
    └─ - ─┘
```

yyy 丸められた後のデータの値の有効数字の 1 から 3 までの 10 進数表示です (*yyy* は、1 ≤ *yyy* < 1000 という整数になるか、出力値がゼロの場合は、*yyy* = 0 となります)。

digit_string

丸められた後のデータの値の *d* 桁の次の有効数字です。

exp 次のいずれかの形式を持つ、3 で割り切れる 10 進表示の指数です (z は 10 進数字です)。

編集記述子	指数の絶対値	指数の形式
EN*w.d*	$\|exp\| \le 99$	E$\pm z_1 z_2$
EN*w.d*	$99 < \|exp\| \le 309$	$\pm z_1 z_2 z_3$
EN*w.d*E*e*	$\|exp\| \le 10^e-1$	E$\pm z_1 \ldots z_e$

数値編集の詳細については、 243 ページの『編集』を参照してください。

EN 編集の例

```
Value         Format      Output
3.14159       EN12.5      b3.14159E+00
1.41425D+5    EN15.5E4    141.42500E+0003
3.14159D-12   EN15.5E1    ***************
```

Fortran 95

```
                  (-qxlf90=signedzero の指定) (-qxlf90=nosignedzero の指定)
-0.001        EN9.2       -1.00E-03                      -1.00E-03
```

Fortran 95 の終り

ES 編集

形式:

ES*w.d*

ES*w.d***E***e*

ES 編集記述子は、出力値がゼロの場合を除いて、有効数字部の絶対値が 1 以上、10 未満である科学計算用数値形式で出力フィールドを作り出します。スケール因数は、出力には影響しません。

ES 編集記述子は、実数タイプの I/O リスト項目、複素数タイプの I/O リスト項目の実数部または虚数部、 IBM または長さが 4 バイト以上の、XL Fortran のそれ以外のタイプに対応します。 IBM

入力フィールドの形式および解釈は、F 編集の場合と同じです。

出力フィールドの形式は、次のとおりです。

```
>>─┬─────┬─┬───┬─.─digit_string─exp──────────────────────────────><
   ├─ + ─┤ └ y ┘
   └─ - ─┘
```

y 丸められた後のデータの値の有効数字の 10 進数表示です。

digit_string
丸められた後のデータの値の *d* 桁の次の有効数字です。

exp 次のいずれかの形式を持つ 10 進表示の指数です (z は 10 進数字です)。

編集記述子	**指数の絶対値**	**指数の形式**
ES*w.d*	$\|exp\| \le 99$	$E\pm z_1 z_2$
ES*w.d*	$99 < \|exp\| \le 309$	$\pm z_1 z_2 z_3$
ES*w.d*E*e*	$\|exp\| \le 10^e-1$	$E\pm z_1 \ldots z_e$

数値編集の詳細については、 243 ページの『編集』を参照してください。

ES 編集の例

```
Value          Format       Output
31415.9        ES12.5       b3.14159E+04
14142.5D+3     ES15.5E4     bb1.41425E+0007
31415.9D-22    ES15.5E1     ***************
```

Fortran 95

```
                   (-qxlf90=signedzero の指定) (-qxlf90=nosignedzero の指定)
-0.001         ES9.2       -1.00E-03                  -1.00E-03
```

Fortran 95 の終り

F (指数なし実数) 編集

形式:

F*w***.***d*

F 編集記述子は、内部形式の実数および複素数と、それを指数なしの文字表現にしたものとの間の編集を指示します。

F 編集記述子は、実数タイプの I/O リスト項目、複素数タイプの I/O リスト項目の実数部または虚数部、 IBM または、長さが 4 バイト以上の、XL Fortran のそれ以外のタイプに対応します。 IBM

F 編集記述子に対応する入力フィールドには、次のものがこの順序でならんでいます。

1. オプションの符号
2. オプションで、小数点を含む数字ストリング。小数点がある場合は、編集記述子に指定されている *d* よりも優先します。小数点が省略されている場合、ストリングの右端の *d* 桁が小数点以下の数と解釈され、必要なら先行ブランクがゼロに変換されます。
3. 次のいずれかの形式のオプションの指数
   - 符号付きの数字ストリング
   - **E**、**D**、または **Q** と、それに続くゼロ個または 1 つ以上のブランク、さらにオプションで符号付き数字ストリングが続きます。 **E**、**D**、および **Q** は、まったく同じように処理されます。

F 編集記述子に対応する入力フィールドは、次のものがこの順序でならんでいます。

1. 必要ならば、ブランク。
2. 内部値が負ならば、負符号。内部値がゼロまたは正ならば、オプションの正符号。
3. 小数点を含んでいる数字ストリング。これは、内部値の絶対値をその時点で有効なスケール因数で修正して、小数点以下の桁数が *d* 桁になるように丸めたものです。詳細については、 266 ページの『P (スケール因数) 編集』を参照してください。

243ページの数値編集についての一般情報を参照してください。

出力の場合、*w* はゼロよりも大きくなければなりません。

Fortran 95

Fortran 95 では、出力の場合、*w* はゼロでもかまいません。 *w* がゼロの場合、出力フィールドは出力値を表すために必要な文字の最小数で構成されます。

Fortran 95 の終り

入力の場合の F 編集の例

(ブランクの解釈には **BN** 編集が有効であると仮定します。)

```
Input      Format     Value
-100       F6.2       -1.0
2.9        F6.2       2.9
4.E+2      F6.2       400.0
```

出力の場合の F 編集の例

```
Value      Format        Output                        Output
                    (-qxlf77=noleadzero の指定)  (-qxlf77=leadzero の指定)
+1.2       F8.4        bb1.2000                      bb1.2000
·12345     F8.3        bbbb.123                      bbbb0.123
-12.34     F6.2        -12.34                        -12.34
```

Fortran 95

```
-12.34     F0.2          -12.34                           -12.34

                   (-qxlf90=signedzero の指定)  (-qxlf90=nosignedzero の指定)
-0.001     F5.2          -0.00                             b0.00
```

Fortran 95 の終り

G (一般) 編集

形式:

G*w*.*d*

G*w*.*d***E***e*

IBM **G***w*.*d***D***e* IBM

IBM **G***w*.*d***Q***e* IBM

G 編集記述子は、任意のタイプの I/O リスト項目に対応します。整数データの編集は、**I** 編集記述子の規則に従い、実数および複素数データの編集は、**E** または **F** 編集記述子の規則に従います (値の大きさによる)。論理データの編集は **L** 編集記述子の規則に従い、文字データの編集は、**A** 編集記述子の規則に従います。

一般化された実数および複素数編集

-qxlf77 オプションの **nogedit77** サブオプション (デフォルト) が指定されると、出力フィールドの表記方法は、編集中のデータの絶対値により決まります。内部データの絶対値を N とします。 $0 < N < 0.1\text{-}0.5\times10^{-d-1}$ または $N \geq 10^{d}\text{-}0.5$ または N が 0 で d が 0 の場合、 **G***w*.*d* の出力編集は *k***PE***w*.*d* の出力編集と同じであり、 **G***w*.*d***E***e* の出力編集は *k***PE***w*.*d***E***e* の出力編集と同じです。ただし *k***P** は、現在有効なスケール因数を指します (266 ページの『P (スケール因数) 編集』も参照)。 もし $0.1\text{-}0.5\times10^{-d-1} \leq N < 10^{d}\text{-}0.5$ または N が 0 で d がゼロでない場合、スケール因数は機能せず、N の値が、次のように編集を決定します。

データの絶対値	等価変換
$N = 0$	F(*w-n*).(*d-1*),*n*('*b*') (d は 0 以外)
$0.1\text{-}0.5\times10^{-d-1} \leq N < 1\text{-}0.5\times10^{-d}$	F(*w-n*).*d*,*n*('*b*')
$1\text{-}0.5\times10^{-d} \leq N < 10\text{-}0.5\times10^{-d+1}$	F(*w-n*).(*d-1*),*n*('*b*')
$10\text{-}0.5\times10^{-d+1} \leq N < 100\text{-}0.5\times10^{-d+2}$	F(*w-n*).(*d-2*),*n*('*b*')
...	...
$10^{d-2}\text{-}0.5\times10^{-2} \leq N < 10^{d-1}\text{-}0.5\times10^{-1}$	F(*w-n*).1,*n*('*b*')
$10^{d-1}\text{-}0.5\times10^{-1} \leq N < 10^{d}\text{-}0.5$	F(*w-n*).0,*n*('*b*')

表中の *b* はブランクです。 *n* は、 **G***w.d* に対して 4 となり、**G***w.d***E***e* に対して *e+2* となります。 *w-n* の値は正の数でなければなりません。

編集対象のデータの絶対値が F 編集の効果的使用を可能にする範囲内でなければ、スケール因数は機能しないことに注意してください。

IBM 拡張

$0 < N < 0.1\text{-}0.5\times10^{-d-1}$、 $N \geq 10^{d}\text{-}0.5$ の場合、または *N* が 0 かつ d が 0 の場合、**G***w.d***D***e* 出力編集は、 *k***PE***w.d***D***e* での出力編集と同じになり、 **G***w.d***Q***e* での出力編集は、*k***PE***w.d***Q***e* での出力編集と同じになります。

IBM 拡張 の終り

出力の場合、**-qxlf77** コンパイラー・オプションの **gedit77** サブオプションが指定されると、数値に応じ **E** または **F** 編集を使ってこの数値が変換されます。フィールドには、必要に応じて、右側にブランクが埋められます。ある数の絶対値を *N* とすれば、編集は次のように行われます。

- $N<0.1$ または $N\geq10^{d}$ の場合:
  - **G***w.d* での編集は、**E***w.d* での編集と同じになります。
  - **G***w.d***E***e* での編集は、 **E***w.d***E***e* での編集と同じになります。
- $N\geq0.1$ および $N<10^{d}$ の場合:

データの絶対値	等価変換
$0.1 \leq N < 1$	F(*w-n*).d, *n*('*b*')
$1 \leq N < 10$	F(*w-n*).(d-1), *n*('*b*')
.	.
.	.
$10^{d-2} \leq N < 10^{d-1}$	F(*w-n*).1, *n*('*b*')
$10^{d-1} \leq N < 10^{d}$	F(*w-n*).0, *n*('*b*')

注: FORTRAN 77 では、値の丸めが出力フィールド形式にどのように影響するかについて扱っていませんが、 Fortran 90 では、その処理が行われます。したがって、**-qxlf77=gedit77** を使用することによって、値と **G** 編集記述子の特定の組み合わせで **-qxlf77=nogedit77** を使用する場合と異なる出力形式が生成される可能性があります。

243ページの数値編集についての一般情報を参照してください。

追加情報は、 243 ページの『編集』を参照してください。

出力における G 編集の例

```
Value       Format         Output                    Output
                   (-qxlf77=gedit77 の指定)   (-qxlf77=nogedit77 の指定)
0.0         G10.2        bb0.00E+00                bbb0.0
0.0995      G10.2        bb0.10E+00                bb0.10
99.5        G10.2        bb100.                    bb0.10E+03
```

I (整数) 編集

形式:

I*w*

I*w*.*m*

I 編集記述子は、内部形式の整数と、整数を文字表現したものとの間の編集を指示します。対応する入出力リスト項目は、整数タイプ、 IBM または XL Fortran の他のタイプになります。 IBM

w はオプションの符号を含みます。

m は、 F95 *w* の値が Fortran 95 でゼロでない限り、 *w* 以下の値を持たなければなりません。 F95

I 編集記述子に対応する入力フィールドは、ブランクだけからなる入力フィールドを除いて、オプションの符号のついた数字ストリングになります。入力フィールドがブランクだけからなる場合、そのような入力フィールドはゼロと見なされます。

m は出力のみ有効です。入力では機能しません。

出力の場合、*w* はゼロよりも大きくなければなりません。

Fortran 95

出力の場合、*w* はゼロでもかまいません。 *w* がゼロの場合、出力フィールドは出力値を表すために必要な文字の最小数で構成されます。

Fortran 95 の終り

I 編集記述子に対応する出力フィールドは、次のものが以下の順序でならんでいます。

1. ゼロ個以上の先行ブランク
2. 内部値が負ならば負符号、ゼロまたは正ならばオプションの正符号
3. 次に示す形式の絶対値
   - *m* を指定していない場合、先行ゼロがない数字ストリング
   - *m* を指定した場合、*m* 桁以上の数字ストリング。必要があれば、先行ゼロが付きます。内部値と *m* が両方ともゼロの場合、ブランクが書き込まれます。

数値編集の詳細については、243ページを参照してください。

m がゼロ、*w* が正の数で、内部データの値がゼロの場合、出力フィールドは、*w* 個のブランク文字で構成されます。 *w* と *m* の両方がゼロで、内部データの値がゼロの場合、出力フィールドは、ブランク文字 1 つのみで構成されます。

入力における I 編集の例

(ブランクの解釈には **BN** 編集が有効であると仮定します。)

```
Input      Format     Value
-123       I6         -123
123456     I7.5       123456
1234       I4         1234
```

出力における I 編集の例

```
Value      Format     Output
-12        I7.6       -000012
12345      I5         12345
```

Fortran 95

```
0          I6.0       bbbbbb
0          I0.0       b
2          I0         2
```

Fortran 95 の終り

L (論理) 編集

形式:

L*w*

L 編集記述子は、内部形式の論理値と、それを文字表現したものとの間の編集を指示します。 **L** 編集記述子は、論理タイプの I/O リスト項目、 IBM または、XL Fortran のそれ以外のタイプに対応します。 IBM

入力フィールドは、オプションのブランクと、それに続くオプションの小数点、さらにそれに続いて真 (true) を表す T または偽 (false) を表す F から構成されます。*w* はブランクを含みます。入力の場合、T または F の後にどのような文字があっても受け入れられますが、後続の文字は無視されます。したがって、 .TRUE. および .FALSE. というストリングは、受け入れ可能な入力形式です。

出力フィールドは、(*w* - 1) 個のブランクと、それに続く T または F から構成されます。

入力における L 編集の例

```
Input       Format      Value
T           L4          true
·FALSE.     L7          false
```

出力における L 編集の例

```
Value       Format      Output
TRUE        L4          bbbT
FALSE       L1          F
```

O (8 進数) 編集

形式:

O*w*

O*w*.*m*

O 編集記述子は、任意のタイプの内部形式の値と、それを 8 進で表現したものとの間の編集を指示します。(8 進数字は、0 から 7 までの数です。)

w はブランクを含みます。

入力の場合、*w* 個の 8 進数字が編集され、入力リスト項目の値の内部表現が作られます。入力フィールド内の 8 進数字は、入力リスト項目に割り当てられた値の内部表現の右寄り部分の 8 進数字に対応します。入力の場合、*m* は機能しません。

出力の場合、*w* はゼロよりも大きくなければなりません。

Fortran 95

出力の場合、*w* はゼロでもかまいません。 *w* がゼロの場合、出力フィールドは出力値を表すために必要な文字の最小数で構成されます。

Fortran 95 の終り

O*w* の出力フィールドは、ゼロ個以上の先行ブランクに続いて、内部値を先行ゼロなしの 8 進数字の形式で表記したもので構成されています。 8 進定数は、常に 1 桁以上からなるという点に注意してください。

O*w.m* の出力フィールドは、数字ストリングが *m* 桁以上であるという点を除いて、**O***w* と同じになります。必要に応じて、数字ストリングには、先行ゼロが埋められます。 *m* の値は、*w* の値がゼロでない限り、 *w* の値を超えてはいけません。*m* がゼロで、内部データの値がゼロの場合、出力フィールドは、符号の制御が有効であるか否かにかかわらず、ブランク文字のみで構成されます。

-qxlf77 コンパイラー・オプションの **nooldboz** サブオプションが指定される場合 (デフォルト)、出力フィールド幅が出力全体を包含するのに不十分であるとき、アスタリス

クが印刷されます。入力の場合、**BN** および **BZ** 編集記述子は、**O** 編集記述子に影響します。

IBM 拡張

-qxlf77 コンパイラー・オプションの **oldboz** サブオプションを指定すると、出力で以下の処理が行われます。

- *m* を *w* の最小値およびデータ項目の最大可能値を表現するために必要な桁数の値と想定すると、 **O***w* は **O***w.m* として扱われます。
- 出力はブランクとそれに続く *m* 桁以上のデータから構成されます。必要に応じて、*m* 桁の数字になるまで右端からゼロが埋められます。数字が大きすぎて出力フィールドに入らないと、右端の *m* 桁分の数字が出力されます。

w がゼロの場合、**oldboz** サブオプションは無視されます。

oldboz サブオプションによって、 **BN** および **BZ** 編集記述子は、**O** 編集記述子に影響しません。

IBM 拡張 の終り

m がゼロ、*w* が正の数で、内部データの値がゼロの場合、出力フィールドは、*w* 個のブランク文字で構成されます。 *w* と *m* の両方がゼロで、内部データの値がゼロの場合、出力フィールドは、ブランク文字 1 つのみで構成されます。

入力における O 編集の例

```
Input      Format      Value
123        O3          83
120        O3          80
```

出力における O 編集の例

```
Value      Format         Output                     Output
                  (-qxlf77=oldboz の指定)  (-qxlf77=nooldboz の指定)
80         O5             00120                      bb120
83         O2             23                         **
```

Fortran 95

```
0          O5.0           bbbbb                      bbbbb
0          O0.0           b                          b
80         O0             120                        120
```

Fortran 95 の終り

Q (文字カウント) 編集

IBM 拡張

形式:
Q

文字カウント **Q** 編集記述子は、入力レコードに残っている文字数を戻します。その結果を使用して残りの入力を制御することができます。

また、拡張精度 **Q** 編集記述子もあります。最初に説明したように、デフォルトでは、XL Fortran は前述の拡張精度 **Q** 編集記述子のみを認識します。詳細については、 247 ページの『E、D、および Q (拡張精度) 編集』を参照してください。両方の **Q** 編集記述子を使用可能にするには、**-qqcount** コンパイラー・オプションを指定する必要があります。詳細については、「ユーザーズ・ガイド」の **-qqcount** を参照してください。

-qqcount コンパイラー・オプションを指定すると、コンパイラーは、**Q** 編集記述子を使用する方法により 2 つの **Q** 編集記述子を区別します。単独の **Q** を検出した場合のみ、コンパイラーは、文字カウント **Q** 編集記述子として解釈します。 **Q***w*. または **Q***w.d* を検出すると、XL Fortran は拡張精度 **Q** 編集記述子として解釈します。正しいセパレーターの正しい形式仕様を使用し、どちらの **Q** 編集記述子を指定したかを XL Fortran に解釈させてください。

文字カウント **Q** 編集記述子の結果として戻された値は、入力レコード長およびそのレコードの現在の文字位置によって異なります。その値は、**FORMAT** ステートメント内の文字カウント **Q** 編集記述子の位置に対応する位置にある **READ** ステートメントのスカラー整変数内へ戻されます。

文字カウント **Q** 編集記述子は、次のファイル・タイプおよびアクセス・モードのレコードを読み取ることができます。
- 定様式順次外部ファイル。このファイル・タイプのレコードは、復帰改行文字で終了します。同じファイル内のレコードの長さはそれぞれ異なります。
- 配列以外の定様式順次内部ファイル。レコード長は、スカラー文字変数の長さとなります。
- 配列の定様式順次内部ファイル。レコード長は、文字配列のエレメントの長さとなります。
- 定様式直接外部ファイル。レコード長は、**OPEN** ステートメント内の **RECL=** 指定子で指定した長さになります。
- 定様式ストリーム外部ファイル。このファイル・タイプのレコードは、復帰改行文字で終了します。同じファイル内のレコードの長さはそれぞれ異なります。

出力操作では、文字カウント **Q** 編集記述子は無視されます。対応する出力項目はスキップされます。

入力の場合の文字カウント Q 編集の例

```
@PROCESS QCOUNT
       CHARACTER(50) BUF
       INTEGER(4) NBYTES
       CHARACTER(60) STRING
       ...
       BUF = 'This string is 29 bytes long.'
       READ( BUF, FMT='(Q)' ) NBYTES
       WRITE( *,* ) NBYTES
! NBYTES equals 50 because the buffer BUF is 50 bytes long.
       READ(*,20) NBYTES, STRING
20     FORMAT(Q,A)
! NBYTES will equal the number of characters entered by the user.
       END
```

IBM 拡張 の終り

Z (16 進) 編集

形式:

Z*w*

Z*w***.***m*

Z 編集記述子は、任意のタイプの内部形式の値とその値を 16 進で表現したものとの間の編集を指示します (16 進数字は、0 ～ 9、A ～ F または a ～ f のいずれかです)。

入力の場合、*w* 個の 16 進数字が編集されて、入力リスト項目の値の内部表現が作られます。入力フィールド内の 16 進数字は、入力リスト項目に割り当てられた値の内部表現の右寄り部分の 16 進数字に対応します。*m* は入力の場合、機能しません。

Fortran 95

出力の場合、*w* はゼロでもかまいません。 *w* がゼロの場合、出力フィールドは出力値を表すために必要な文字の最小数で構成されます。

Fortran 95 の終り

Z*w* の出力フィールドは、ゼロ個以上の先行ブランクに続いて、内部値を先行ゼロなしの 16 進数字の形式で表記したもので構成されています。 16 進定数は、常に 1 桁以上からなるという点に注意してください。

Z*w.m* の出力フィールドは、数字ストリングが *m* 桁以上であるという点を除いて、**Z***w* と同じになります。必要に応じて、数字ストリングには、先行ゼロが埋められます。 *m* の値は、F95 *w* の値がゼロでない限り、*w* の値を超えてはいけません。F95 *m* がゼロで、内部データの値がゼロの場合、出力フィールドは、符号の制御が有効であるか否かにかかわらず、ブランク文字だけで構成されます。

m がゼロ、*w* が正の数で、内部データの値がゼロの場合、出力フィールドは、*w* 個のブランク文字で構成されます。

Fortran 95

w と *m* の両方がゼロで、内部データの値がゼロの場合、出力フィールドは、ブランク文字 1 つのみで構成されます。

Fortran 95 の終り

-qxlf77 コンパイラー・オプションの **nooldboz** サブオプションが指定される場合 (デフォルト)、出力フィールド幅が出力全体を包含するのに不十分であるとき、アスタリスクが印刷されます。入力の場合、**BN** および **BZ** 編集記述子は、**Z** 編集記述子に影響します。

IBM 拡張

-qxlf77 コンパイラー・オプションの **oldboz** サブオプションを指定すると、出力で以下の処理が行われます。

- *m* を *w* の最小値およびデータ項目の最大可能値を表現するために必要な桁数の値と想定すると、 **Z***w* は **Z***w.m* として扱われます。
- 出力はブランクとそれに続く *m* 桁以上のデータから構成されます。必要に応じて、*m* 桁の数字になるまで右端からゼロが埋められます。数字が大きすぎて出力フィールドに入らないと、右端の *m* 桁分の数字が出力されます。

w がゼロの場合、**oldboz** サブオプションは無視されます。

oldboz サブオプションによって、 **BN** および **BZ** 編集記述子は、 **Z** 編集記述子に影響しません。

IBM 拡張 の終り

入力の場合の Z 編集の例

```
Input      Format     Value
0C         Z2         12
7FFF       Z4         32767
```

出力の場合の Z 編集の例

```
Value      Format          Output                    Output
                 (-qxlf77=oldboz の指定)  (-qxlf77=nooldboz の指定)
-1         Z2              FF                        **
12         Z4              000C                      bbbC
```

Fortran 95

```
12          Z0             C                       C
0           Z5.0           bbbbb                   bbbbb
0           Z0.0           b                       b
```

Fortran 95 の終り

制御編集記述子

/ (スラッシュ) 編集

形式:

```
/
r/
```

スラッシュ編集記述子は、現在のレコードに関するデータ転送の終わりを示します。繰り返し指定子 (*r*) は、デフォルト値として 1 を持ちます。

順次アクセスを使用して入力用のファイルに接続する場合、スラッシュ編集記述子を検出するたびに、ファイルは次のレコードの始めに位置付けられます。

順次アクセスを使用して出力用のファイルに接続する場合、スラッシュ編集記述子を検出するたびに、新しいレコードが作成され、ファイルがその新しいレコードの開始点から書き込まれるように位置付けられます。

直接アクセスを使用して入力または出力用のファイルに接続する場合、スラッシュ編集記述子を検出するたびに、レコード番号が 1 つずつ増やされ、そのレコード番号のついたレコードの先頭にファイルが位置付けられます。

ストリーム・アクセスを使用して入力用のファイルに接続する場合、スラッシュ編集記述子を検出するたびに、ファイルは次のレコードの先頭に位置付けられ、現在のレコードの残りの部分がスキップされます。ストリーム・アクセス用に接続されたファイルへの出力時は、新たに作成された空のレコードが現在のレコードの後に続きます。新しいレコードは現在のレコードおよび、ファイルの最後のレコードの両方になり、新しいレコードの先頭がファイル位置になります。

入力の場合のスラッシュ編集の例

```
500   FORMAT(F6.2 / 2F6.2)
100   FORMAT(3/)
```

: (コロン) 編集

形式:

```
:
```

I/O リストにそれ以上項目がない場合、コロン編集記述子は、形式制御を終了させます。コロンが検出されたときに、I/O リスト内にまだ項目が残っている場合、コロンは無視されます。詳細については、 269 ページの『I/O リストと形式仕様の相互作用』 を参照してください。

コロン編集の例

```
10    FORMAT(3(:'Array Value',F10.5)/)
```

$ (ドル記号) 編集

IBM 拡張

形式:

```
$
```

ドル編集記述子は、順次または定様式ストリーム **WRITE** ステートメントのレコード終わり処理を抑止します。通常、形式仕様の終わりに達すると、現在のレコードのデータ伝送が停止し、ファイルは、次の I/O 操作によって新しいレコードが処理されるように位置付けられます。しかし、形式仕様内にドル記号がある場合は、自動的なレコードの終わり処理は抑止されます。それ以降の I/O ステートメントで、同じレコードに関する読み取りまたは書き込みを行うことができます。

ドル記号編集の例

ドル記号編集は、通常、応答の指示や、同じ行からの応答の読み取りを行うために使用します。

```
      WRITE(*,FMT='($,A)')'Enter your age  '
      READ(*,FMT='(BN,I3)')IAGE
      WRITE(*,FMT=1000)
1000  FORMAT('Enter your height:  ',$)
      READ(*,FMT='(F6.2)')HEIGHT
```

IBM 拡張 の終り

アポストロフィ / 二重引用符編集 (文字ストリング編集記述子)

形式:

'*文字ストリング*'
"*character string*"

アポストロフィ / 二重引用符の編集記述子は、出力形式仕様内に文字リテラル定数を指定します。出力フィールドの幅は、文字リテラル定数の長さになります。文字リテラル

定数の詳細な内容については、 37ページを参照してください。

IBM 拡張

注:

1. バックスラッシュは、デフォルトではエスケープ・シーケンスとして認識され、 **-qnoescape** コンパイラー・オプションが指定されている場合はバックスラッシュ文字として認識されます。エスケープ・シーケンスに関する詳細については、 38ページを参照してください。
2. XL Fortran では、文字定数、ホレリス定数、文字ストリング編集記述子、および注釈の中でのマルチバイト文字をサポートします。このサポートは、**-qmbcs** オプションによって提供されます。ストリング全体を保持するには短すぎる変数にマルチバイト文字が入っている定数を割り当てると、マルチバイト文字の内部で切り捨てが起こることがあります。
3. Unicode の文字およびファイル名もサポートします。 環境変数 **LANG** が **UNIVERSAL** に設定され、**-qmbcs** コンパイラー・オプションが指定されている場合、コンパイラーは Unicode の文字およびファイル名の読み取りや書き込みを行うことができます。

IBM 拡張 の終り

アポストロフィ / 二重引用符編集の例

```
      ITIME=8
      WRITE(*,5) ITIME
5     FORMAT('The value is -- ',I2)           ! The value is -- 8
      WRITE(*,10) ITIME
10    FORMAT(I2,'o''clock')                  ! 8o'clock
      WRITE(*,'(I2,''o''''clock'')') ITIME    ! 8o'clock
      WRITE(*,15) ITIME
15    FORMAT("The value is -- ",I2)           ! The value is -- 8
      WRITE(*,20) ITIME
20    FORMAT(I2,"o'clock")                    ! 8o'clock
      WRITE(*,'(I2,"o''clock")') ITIME        ! 8o'clock
```

BN (ブランク・ヌル) および BZ (ブランク・ゼロ) 編集

形式:

BN

BZ

BN および **BZ** 編集記述子は、引き続いて処理されると、**I**、**F**、**E**、**EN**、**ES**、**D**、**G**、**B**、**O**、 **Z**、および拡張精度 **Q** 編集記述子に、非先行ブランクをどのように解釈させるかを制御します。**BN** および **BZ** は入力の場合にのみ機能します。

BN は、数字入力フィールド内のブランクを無視することを指定し、残りの文字は右寄せされる場合と同様に解釈されます。ブランクだけからなるフィールドの値はゼロとなります。

BZ は、数字入力フィールド内の非先行ブランクをゼロとして解釈するように指定します。

ブランクの解釈の最初の設定は、**OPEN** ステートメントの **BLANK=** 指定子により決まります。 (420 ページの『OPEN』を参照してください。) 最初の設定は、次のように決まります。
- **BLANK=** を指定していない場合、ブランクの解釈は、**BN** 編集を指定した場合と同じです。
- **BLANK=** を指定している場合に、指定子の値が **NULL** であると、ブランクの解釈は、**BN** 編集を指定した場合と同様になりますが、指定子の値が **ZERO** であると、**BZ** 編集を指定した場合と同様になります。

ブランクの解釈について、最初に設定された内容は、定様式の **READ** ステートメントの実行開始点から、**BN** または **BZ** 編集記述子を検出するか、または形式制御が終了するまで有効です。 **BN** または **BZ** 編集記述子を検出すると、別の **BN** または **BZ** 編集記述子を検出するか、形式制御が終了するまで、その設定は有効となります。

IBM 拡張

-qxlf77 コンパイラー・オプションの **oldboz** サブオプションを指定すると、**BN** および **BZ** 編集記述子は、**B**、**O**、または **Z** 編集記述子で編集されたデータ入力に影響しません。ブランクは、ゼロとして解釈されます。

IBM 拡張 の終り

H 編集

形式:

n **H** *str*

H 編集記述子は、出力形式仕様内で文字ストリング (*str*) およびその長さ (*n*) を指定します。ストリングは、文字リテラル定数に使用できる任意の文字で構成されます。

H 編集記述子が文字リテラル定数内にあり、定数区切り文字 (たとえば、アポストロフィ) が 2 つ続いている場合、それらの文字は *str* 内に表されます。それ以外の場合は、別の区切り文字を使用する必要があります。

H 編集記述子は、入力では使用できません。

注:

IBM 拡張

1. バックスラッシュは、デフォルトではエスケープ文字として認識され、**-qnoescape** コンパイラー・オプションが指定されている場合はバックスラッシュ文字として認識されます。エスケープ・シーケンスに関する詳細については、 38ページを参照してください。
2. XL Fortran では、文字定数、ホレリス定数、文字ストリング編集記述子、および注釈の中でのマルチバイト文字をサポートします。このサポートは、**-qmbcs** オプションによって提供されます。ストリング全体を保持するには短すぎる変数にマルチバイト文字が入っている定数を割り当てると、マルチバイト文字の内部で切り捨てが起こることがあります。
3. Unicode の文字およびファイル名もサポートします。 環境変数 **LANG** が **UNIVERSAL** に設定され、**-qmbcs** コンパイラー・オプションが指定されている場合、コンパイラーは Unicode の文字およびファイル名の読み取りや書き込みを行うことができます。

IBM 拡張 の終り

Fortran 95

4. **H** 編集記述子は FORTRAN 77 と Fortran 90 の一部でしたが、Fortran 95 には組み込まれていません。詳細については、 927 ページの『削除された機能』を参照してください。

Fortran 95 の終り

H 編集の例

```
50    FORMAT(16HThe value is -- ,I2)
10    FORMAT(I2,7Ho'clock)
      WRITE(*,'(I2,7Ho''clock)') ITIME
```

P (スケール因数) 編集

形式:

*k***P**

指定したスケール因数 *K* は、別のスケール因数を検出するか、形式制御が終了するまで、後で処理されるすべての **F**、**E**、**EN**、**ES**、**D**、**G**、および拡張精度 **Q** 編集記述子に適用されます。各 I/O ステートメントの実行が開始される時点では、*k* の値はゼロです。 k の値は、10 の累乗を表す整数値で、必要な場合には符号が付きます。

入力の場合、**F**、**E**、**EN**、 **ES**、**D**、**G**、または拡張精度 **Q** 編集記述子を指定した入力フィールドに指数が入っていると、そのスケール因数は無視されます。指数が入っていなければ、内部値は、外部値に $10^{(-k)}$ を掛けた値に等しくなります。

出力の場合:

- **F** 編集では、外部値は内部値に 10^k を掛けた値に等しくなります。
- **E**、**D**、および拡張精度 **Q** 編集では、外部 10 進フィールドに 10^k が乗じられます。指数は k を引いた数になります。
- **G** 編集では、**F** 編集を使用できる範囲内にフィールドがある限り、フィールドはスケール因数の影響を受けません。 **E** 編集が必要な場合、スケール因数は、**E** 出力編集を使用した場合と同様に機能します。
- **EN** および **ES** 編集では、スケール因数は機能しません。

入力における P 編集の例

```
Input       Format      Value
98.765      3P,F8.6     .98765E-1
98.765      -3P,F8.6    98765.
.98765E+2   3P,F10.5    .98765E+2
```

出力における P 編集の例

```
Value       Format          Output                    Output
                     (-qxlf77=noleadzero の指定)  (-qxlf77=leadzero の指定)
5.67        -3P,F7.2        bbbb.01                   bbb0.01
12.34       -2P,F6.4        b.1234                    0.1234
12.34       2P,E10.3        b12.34E+00                b12.34E+00
```

S、SP、および SS (符号制御) 編集

形式:

S
SP
SS

S、**SP**、および **SS** 編集記述子は、後で処理されるすべての **I**、**F**、**E**、 **EN**、**ES**、**D**、**G**、および拡張精度 **Q** 編集記述子の正符号を制御します。これは、別の **S**、**SP**、または **SS** 編集記述子を検出するか、あるいは形式制御が終了するまで有効です。

S および **SS** は、正符号を書き込まないという指定です。 (どちらの編集記述子を指定しても、結果は同じになります。) **SP** は、正符号を書き込む指定です。

出力における S、SS、および SP 編集の例

```
Value       Format      Output
12.3456     S,F8.4      b12.3456
12.3456     SS,F8.4     b12.3456
12.3456     SP,F8.4     +12.3456
```

T、TL、TR、および X (定位置) 編集

形式:

T*c*
TL*c*
TR*c*
*o***X**

T、**TL**、**TR**、および **X** 編集記述子は、次に行う文字の転送を、レコード内のどの位置から開始するかを指定します。 **T** および **TL** 編集記述子は、ファイル位置で左タブ制限を使用します。データ転送の直前、左タブ制限の定義は、ストリーム・ファイルの現在のレコードまたは現在の位置の文字位置になります。 **T**、**TL**、**TR**、および **X** は文字位置を次のように指定します。

- **T***c* の場合、左タブ制限に対して *c* 番目の文字位置です。
- **TL***c* の場合、現在位置の左側の *c* 番目です。ただし、*c* が現行位置と左タブ制限の差よりも大きい場合を除きます。レコードからの、またはレコードへの次の文字の伝送は、左タブ制限で行われます。
- **TR***c* の場合、現在位置の右側の *c* 番目の文字位置です。
- *o* **X** の場合、現在位置の右側の *o* 番目の文字位置です。

TR および **X** 編集記述子の結果は同じです。

入力の場合、**TR** または **X** 編集記述子によって、レコードの最後の文字を超えた位置を指定することができます。

出力の場合、**T**、**TL**、**TR**、または **X** 編集記述子だけでは、文字は転送されません。この編集記述子で指定した位置以降で文字を転送する場合、スキップした位置および以前に埋められていない位置はブランクで埋められます。結果は、そのレコード全体が最初からブランクであった場合と同じです。

出力の場合、**T**、**TL**、**TR**、または **X** 編集記述子によって、位置が指定し直されます。その結果、後から他の編集記述子を使用して編集を行うと、文字が置換されます。

IBM 拡張

X 編集記述子は、文字位置なしで指定することができます。これは、`1X` として扱われます。ソース・ファイルを **-qlanglvl=90std** または **-qlanglvl=95std** でコンパイルすると、この拡張機能はコンパイル時形式仕様すべてで使用不能とされ、 *o***X** が実行されます。 この拡張機能を実行時形式で使用不能にするには、次の実行時オプションを設定する必要があります。

```
XLFRTEOPTS="langlvl=90std" or "langlvl=95std" ; export XLFRTEOPTS
```

IBM 拡張 の終り

入力における T、TL、および X 編集の例

```
150   FORMAT(I4,T30,I4)
200   FORMAT(F6.2,5X,5(I4,TL4))
```

出力における T、TL、TR、および X 編集の例

```
50    FORMAT('Column 1',5X,'Column 14',TR2,'Column 25')
100   FORMAT('aaaaa',TL2,'bbbbb',5X,'ccccc',T10,'ddddd')
```

I/O リストと形式仕様の相互作用

形式指示の形式設定では、まず、形式制御が開始されます。形式制御の各処理は、形式仕様内で次に指定されている編集記述子、および I/O リスト内で次に指定されている項目 (もしあれば) によって決まります。

I/O リストに項目が 1 つ以上指定されていれば、形式仕様にも、データ編集記述子が 1 つ以上必要です。空の形式仕様 (括弧のみ) を使用できるのは、I/O リスト内に項目がない場合か、または各項目がゼロ・サイズ配列である場合に限られることに注意してください。このような場合に、アドバンス I/O が有効であると、入力レコードが 1 つスキップされるか、または文字を含まない出力レコードが 1 つ書き込まれます。非アドバンス I/O の場合には、ファイルの位置は変更されません。

形式仕様は、繰り返し仕様 (*r*) がある場合を除いて、左から右に向かって解釈されます。繰り返し仕様が前に付いている形式項目は、繰り返し仕様が付いていない形式仕様または編集記述子と同様の形式仕様あるいは編集記述子が *r* 個あるリストとして処理されます。

I/O リストで指定される 1 つの項目は、各データ編集記述子に対応します。複素数タイプの各リスト項目には、 2 個の **F**、**E**、**EN**、**ES**、 **D**、**G**、または拡張精度 **Q** 編集記述子の解釈が必要です。各制御編集記述子または文字ストリング編集記述子に対しては、 I/O リストで指定される項目は対応しません。形式制御は、レコードと直接に情報を交換します。

形式制御は、次のように実行されます。

1. データ編集記述子を検出すると、形式制御は I/O リスト内に項目があればその項目を処理し、なければ I/O コマンドを終了させます。処理するリスト項目が複素数タイプの場合、いずれか 2 つの編集記述子を処理します。
2. I/O リストにそれ以上項目がない場合、コロン編集記述子は、 形式制御を終了させます。コロンが検出されたときに、I/O リスト内にまだ項目が残っている場合はコロンは無視されます。
3. 形式仕様の終わりに達すると、I/O リスト全体が処理されていれば、形式制御が終了し、それ以外の場合は、右括弧のすぐ手前で終わっている形式項目の始めに戻ります。後者の場合、次のような項目が適用されます。

- 形式仕様の再使用部分には、データ編集記述子が少なくとも 1 つ含まれていなければなりません。
- 繰り返し仕様のすぐ後にある括弧に戻る場合は、その繰り返し仕様が再度使用されます。
- 戻ること自体は、スケール因数や **S**、**SP**、または **SS** 編集記述子、あるいは **BN** または **BZ** 編集記述子に影響を与えません。
- 形式制御が戻るとき、ファイルはスラッシュ編集記述子が処理されるバックアップ値と同じ方法で位置付けられます。

IBM 拡張

読み取り操作では、レコード内の未処理文字は、次のレコードが読み取られるときにすべてスキップされます。形式指示の形式設定で処理される入力レコード内の非文字データ値のセパレーターとして、コンマを使用することができます。コンマがフィールド幅の最後より前にある場合、形式幅仕様をオーバーライドします。たとえば、(I10,F20.10,I4) という形式は、以下のレコードを正しく読み取ります。

```
-345, .05E-3, 12
```

IBM 拡張 の終り

FORMAT ステートメントで Fortran レコードを定義する場合、使用する I/O メディアで許可される最大サイズのレコードを考慮に入れることが重要です。たとえば Fortran レコードを印刷する場合、レコードの長さはプリンターの行の長さ以下でなければなりません。

リスト指示の形式設定

リスト指示の形式設定の場合、編集は、読み取りまたは書き込み対象のデータのタイプおよび長さによって制御されます。アスタリスクの形式識別子が、リスト指示の形式設定を指定します。たとえば、次のようになります。

```
REAL TOTAL1, TOTAL2
PRINT *, TOTAL1, TOTAL2
```

リスト指示の形式設定を使用できるのは、順次およびストリーム・アクセスの場合に限られます。

リスト指示形式設定によって処理される定様式レコード内の文字は、値のセパレーターで分けられた、次のような一連の値を構成します。

- 値の形式は、定数値またはヌル値です。
- 値のセパレーターは、コンマ、スラッシュ、またはブランクです。コンマやスラッシュの前後には、1 つ以上のブランクを入れることができます。

リスト指示入力

リスト指示の **READ** ステートメント内の入力リスト項目は、レコード内の対応する値によって定義されます。各入力値の形式は、入力リスト項目のタイプとして許されるものにしてください。入力値は、次のいずれかです。

- *c*
- *r* * *c*
- *r* *

c は、組み込みタイプのリテラル定数または区切りのない文字定数のいずれかです。 *r* は無符号でゼロ以外の整数のリテラル定数です。 kind 型付きパラメーターは、*r* または *c* のいずれかに対して指定しなければなりません。定数 *c* は、対応するリスト項目と同じ kind 型付きパラメーターを持つものと解釈されます。

r * *c* の形式は定数を *r* 個続けて指定するのと同じです。 *r* * の形式はヌル値を *r* 個続けて指定するのと同じです。

ヌル値は、次のように表されます。

- 2 個続けたコンマ。途中にブランクが入る場合もあります。
- コンマの後のスラッシュ。途中にブランクが入る場合もあります。
- レコード内の最初のコンマ。このコンマの前にブランクが入る場合もあります。

IBM 拡張

-qintlog コンパイラー・オプションを使用して、整数または論理タイプのいずれかの入力項目の整数値または論理値を指定します。

IBM 拡張 の終り

文字値を、必要な個数のレコードに続けることができます。次の有効な項目が文字タイプで、以下の条件を満たす場合、区切り文字となるアポストロフィおよび引用符は必要ありません。

1. 文字定数が、値のセパレーターであるブランク、コンマ、またはスラッシュを含まない。
2. 文字定数が、レコードの境界にまたがらない。
3. 最初の非ブランク文字が、引用符またはアポストロフィでない。
4. 先行文字が、アスタリスクが後に続く数値ではない。
5. 文字定数が 1 つ以上の文字を含む。

区切り文字が省略されると、文字定数は最初のブランク、コンマ、スラッシュ、またはレコードの終わりによって終了され、データ内のアポストロフィおよび二重引用符は、二重にされません。

レコードの終わりには、次のような効力があります。
- ブランクが文字リテラルまたは複素数リテラル定数内にない場合のブランク・セパレーターと同じ効力があります。
- 文字値にブランクまたは他の文字を挿入できなくなります。
- 1 つのアポストロフィを表すのに使われている 2 つのアポストロフィの間を分けることはできません。

2 つ以上の連続したブランクは、文字値内にある場合を除いて、1 つのブランクとして処理されます。

ヌル値は、対応する入力リスト項目の定義状況に影響を与えません。

スラッシュは入力リストの終わりを示し、リスト指示の形式設定を終了します。スラッシュが検出されたときに、入力リストに追加の項目が残っている場合、それらの項目にヌル値が指定された場合と同じように処理されます。

入力リスト内に派生型のオブジェクトがあると、そのオブジェクトは、構造体コンポーネントのすべてが派生型定義の順序でリストされている場合と同様に処理されます。派生型の最終コンポーネントは、ポインターまたは割り当て可能なものであってはなりません。

リスト指示出力

リスト指示の **WRITE** および **PRINT** ステートメントは、入力時と同じ順序で、出力リストに値を出力します。値は、各出力リスト項目のデータ型に合った形式で書き込まれます。

複素定数および文字定数を除いて、レコードの終わりが、定数内にあってはなりません。また、ブランクを定数内に入れることもできません。

整数値は、**I** 編集を使って書き込まれます。

実数値は、**E** または **F** 編集を使って書き込まれます。 (詳細については、 247 ページの『E、D、および Q (拡張精度) 編集』 または 251 ページの『F (指数なし実数) 編集』 を参照してください。)

複素定数は、その実数部と虚数部がコンマで区切られて括弧で囲まれ、それぞれが、実定数について、上で定義したように出力されます。定数全体の長さがレコード全体の長さ以上である場合にのみ、レコードの終わりは、コンマと虚数部の間にあります。複素定数内で埋め込まれたブランクが許可されるのは、コンマとレコードの終わりの間の 1 ブランク、および次のレコードの始めの 1 ブランクのみです。

論理値は、真の値は `T`、偽の値は `F` と書き込まれます。

内部ファイル用、**DELIM=** 指定子なしでオープンされるファイル用、または **DELIM=** 指定子に値 **NONE** を指定してオープンされるファイル用の文字定数は次のようになります。

- アポストロフィや引用符によって区切られません。
- 値のセパレーターによって区切られません。
- 1 つのアポストロフィまたは二重引用符によって外部的に表現される内部的なアポストロフィまたは二重引用符をそれぞれ持ちます。
- 先行するレコードから継続している文字定数で始まるレコードの先頭で紙送り制御を行うためにプロセッサーによって挿入されたブランク文字を持ちます。

区切りのない文字データは、リスト指示入力を使っても、正しく読み返すことができない場合があります。

値が **QUOTE** で **DELIM=** 指定子付きでオープンされたファイル用に出力された文字定数は、二重引用符によって区切られ、その後に値のセパレーターが続き、2 つの連続する二重引用符によって外部メディア上に表される内部引用符を持ちます。値が **APOSTROPHE** で **DELIM=** 指定子付きでオープンされたファイル用に出力された文字定数は、アポストロフィによって区切られ、その後に値のセパレーターが続き、2 つの連続するアポストロフィによって外部メディア上に表される内部アポストロフィを持ちます。

スラッシュ (値のセパレーターとして使われる) およびヌル値は書き込まれません。

配列は順序で書き込まれます。

出力リスト内に、構造体を指定することができます。リスト指示出力では、構造体は、そのコンポーネントのすべてが派生型の定義に定義されているのと同じ順序でリストされているものとして扱われます。派生型の最終コンポーネントは、ポインターまたは割り当て可能なものであってはなりません。

IBM 拡張

以下の表は、書き込まれるフィールドの幅を、データ型と長さ別に示しています。レコードの大きさは、フィールド幅の合計に各非文字フィールドを区切る 1 バイトを加えたものになります。

表 13. 書き込みフィールドの幅

データ型	長さ (バイト)	フィールドの最大幅 (文字数)	小数 (10 進数)	精度/IEEE (10 進数)
整数	1	4	n/a	n/a
	2	6	n/a	n/a
	4	11	n/a	n/a
	8	20	n/a	n/a

表 13. 書き込みフィールドの幅 (続き)

データ型	長さ (バイト)	フィールドの最大幅 (文字数)	小数 (10 進数)	精度/IEEE (10 進数)
実数	4 8 16	17 26 43	10 18 35	7 15 31
複素数	8 16 32	37 55 89	10 18 35	7 15 31
論理	1 2 4 8	1 1 1 1	n/a n/a n/a n/a	n/a n/a n/a n/a
文字	n	n	n/a	n/a

IBM 拡張 の終り

区切られた文字定数が継続する場合を除いて、各出力レコードは、レコードを印刷するときに紙送り制御を行うために、ブランク文字で始まります。

名前リストの形式設定

Fortran 90 では、名前リスト形式は、順次アクセスでのみ使用できます。

IBM 拡張

XL Fortran では、名前リストの形式設定を内部ファイルおよびストリーム・アクセスで使用することもできます。

IBM 拡張 の終り

名前リスト入力データ

名前リスト入力の入力形式は次のとおりです。

1. オプションのブランク。
2. アンパーサンド (&) 文字と、その直後に **NAMELIST** ステートメント内で指定した名前リストグループ名が続きます。
3. 1 つ以上のブランク。
4. 値セパレーターで区切られた、ゼロ個以上の一連の名前値サブシーケンス。
5. 名前リスト入力を終了されるスラッシュ。

区切られた文字定数を継続する入力レコードの先頭のブランクは、定数の一部と見なされます。

IBM 拡張

NAMELIST 実行時オプションが **OLD** の場合、 **NAMELIST** ステートメントに対する入力は、次のものからなります。

1. オプションのブランク。
2. アンパーサンド (&) またはドル記号 ($)、その直後に **NAMELIST** ステートメント内で指定した名前リスト・グループ名が続きます。
3. 1 つ以上のブランク。
4. 1 つのコンマで区切られた、ゼロ個以上の一連の名前値サブシーケンス。コンマは、最後の名前値サブシーケンスの後に指定できます。
5. データ・グループの終わりを示す `&END` または `$END`。
6. 各入力レコードの最初の文字は、ブランクになります。この中には、区切られた文字定数を継続させるレコードも含まれます。

IBM 拡張 の終り

Fortran 95

Fortran 95 では、名前リスト内で注釈を使うことができます。

NAMELIST 実行時オプションに **NEW** または **OLD** のどちらの値が指定されているかに応じて、異なる規則が適用されます。

値 **NEW** が **NAMELIST** 実行時オプションに指定されている場合は、名前リストの注釈には以下の規則が適用されます。

- 文字リテラル定数内の場合を除き、スラッシュ以外の値のセパレーターの後の感嘆符 (!)、または名前リスト入力レコードの最初の非ブランク位置にある感嘆符 (!) によって注釈は開始されます。
- 注釈はその入力レコードの終わりまで続き、プロセッサー依存の文字セット内にあるどの文字でも入力可能です。
- 注釈は無視されます。
- 名前リストの注釈内のスラッシュでは、名前リスト入力ステートメントの実行は終了しません。

Fortran 95 の終り

IBM 拡張

値 **OLD** が **NAMELIST** 実行時オプションに指定されている場合は、名前リストの注釈には以下の規則が適用されます。

- 文字リテラル定数内の場合を除き、単一のコンマの後の感嘆符 (!)、または名前リスト入力レコードの最初の非ブランク位置にある感嘆符 (!) (ただし、入力レコードの最初の文字ではない) によって注釈は開始されます。
- 注釈はその入力レコードの終わりまで続き、プロセッサー依存の文字セット内にあるどの文字でも入力可能です。
- 注釈は無視されます。
- 名前リストの注釈内の &END または $END では、名前リスト入力ステートメントの実行は終了しません。

IBM 拡張 の終り

名前リストの注釈はストリーム入力では使用できません。

入力レコード内の名前値サブシーケンスの形式は次のとおりです。

```
►►──name── = ──constant_list──────────────────────────────────────────────►◄
```

name 変数です。

constant 次の形式をとります。

```
►►──┬────┬──literal_constant──────────────────────────────────────────────►◄
    └─r*─┘
```

r *literal_constant* が発生する回数を指定するゼロ以外の無符号のスカラー、整数リテラル定数です。 *r* には、kind タイプ・パラメーターは指定できません。

literal_constant
kind タイプ・パラメーターを指定できない組み込みタイプのスカラー・リテラル定数であるか、またはヌル値です。この定数は、対応するリスト項目と同じ kind タイプ・パラメーターを持つ場合と同じように扱われます。 *literal_constant* が文字タイプの場合、アポストロフィまたは引用符で区切る必要があります。 *literal_constant* が論理タイプの場合、T または F として指定することができます。

name を修飾するために使用した添え字、ストライド、およびサブストリングの範囲式はすべて、kind タイプ・パラメーターが指定されていない整数リテラル定数となります。

文字以外の入力データ型に関する内容については、 271 ページの『リスト指示入力』を参照してください。

name が配列でものオブジェクトでもない場合、 *constant_list* には単一の定数しか入れられません。

入力ファイルに指定されている変数名を、名前リストに指定しなければなりませんが、入力データの順序は自由です。 *name* と等しくなるように設定した名前で、名前リスト内の名前を代替することはできません。名前リストに指定できるものの詳細については、 418 ページの『NAMELIST』を参照してください。

各名前値サブシーケンスにおいて、名前はオプションの修飾子付きの名前リスト・グループ項目の名前でなければなりません。オプションの修飾子付きの名前は、ゼロ・サイズ配列、ゼロ・サイズ配列セクション、またはゼロ長の文字ストリングになることはできません。オプションである修飾子が指定された場合には、ベクトル添え字を入れることはできません。

name がベクトル添え字なしの配列または配列セクションの場合は、格納された順に、配列の全エレメントのリストに展開されます。 *name* が構造体の場合、定義で指定された順序にしたがって、組み込みタイプの最終コンポーネントのリストに展開されます。派生型の最終コンポーネントは、ポインターまたは割り当て可能なものであってはなりません。

name が配列または構造体の場合、*constant_list* 内の定数の数は、*name* の拡張で指定した項目数以下になります。定数の数が項目数を下回る場合は、残りの項目は前の値を保持します。

ヌル値は、次のように指定します。
- *r* * 形式
- 等号の後に続く 2 つの連続する値のセパレーターの間のブランク
- 等号の後、最初の値のセパレーターの前の、ゼロ個以上のブランク
- 2 つの連続する非ブランク値のセパレーター

ヌル値は、対応する入力リスト項目の定義状況に影響を与えません。 名前リスト・グループ・オブジェクト・リスト項目が定義されると、前の値を保持します。定義されない場合は、未定義状態のままになります。ヌル値を複素数の実数部または虚数部として使用することはできませんが、単一のヌル値は、複素定数全体を表すことができます。

値のセパレーターの後に続くレコードの終わりは、ブランクが途中に入るか否かにかかわらず、ヌル値を指定しません。

IBM 拡張

LANGLVL 実行時オプションが **EXTENDED** に設定されると、XL Fortran によって複数の入力値を単一の配列エレメントとともに指定することができます。配列エレメント

で、サブオブジェクトの指定子を指定することはできません。これが行われると、値は配列エレメントの順に、配列の連続するエレメントに割り当てられます。たとえば、配列 A が次のように宣言されたと想定します。

```
      INTEGER A(100)
      NAMELIST /FOO/ A
      READ (5, FOO)
```

続いて、次の入力が装置 5 に指定されます。

```
 &FOO
 A(3) = 2, 10, 15, 16
 /
```

READ ステートメントの実行時に、A(3) に値 2、A(4) に値 10、 A(5) に値 15、A(6) に値 16 がそれぞれ割り当てられます。

複数の値を単一の配列エレメントとともに指定する場合は、論理定数の前にピリオドを付けて指定しなければなりません (例: .T)。

NAMELIST 実行時オプションが値 **OLD** を用いて指定されている場合、**BLANK=** 指定子は、非文字定数の間にある埋め込まれたブランクおよび後書きブランクをどのように扱うかを指定します。

-qmixed コンパイラー・オプションを指定すると、名前リスト・グループ名およびリスト項目名で、大文字小文字の区別が行われます。

IBM 拡張 の終り

名前リスト入力ステートメントの実行時に値のセパレーターとしてスラッシュが検出されると、以前の値の割り当てが行われた後にその入力ステートメントの実行が終了されます。転送中の名前リスト・グループ・オブジェクト内に追加項目がある場合、ヌル値が入れられたのと同様の効力を持ちます。

名前リスト入力データの例

READ ステートメントが実行される前には、ファイル NMLEXP には次のデータが入っています。

文字位置:

```
        1         2         3
1...+....0....+....0....+....0
```

ファイルの内容:

```
 &NAME1
 I=5,
 SMITH%P_AGE=40
 /
```

上記のファイルには、4 つのデータ・レコードが入っています。プログラムには、次のデータが入っています。

```
TYPE PERSON
  INTEGER P_AGE
  CHARACTER(20) P_NAME
END TYPE PERSON
TYPE(PERSON) SMITH
NAMELIST /NAME1/ I,J,K,SMITH
I=1
J=2
K=3
SMITH=PERSON(20,'John Smith')
OPEN(7,FILE='NMLEXP')
READ(7,NML=NAME1)
! Only the value of I and P_AGE in SMITH are
! altered (I = 5, SMITH%P_AGE = 40).
! J, K and P_NAME in SMITH remain the same.
END
```

注: 前の例では、データ項目は別のデータ・レコードに指定されています。以下の例は、同一のデータ項目を格納しているファイルの例ですが、1 つのデータ・レコードになっています。

文字位置:

```
        1         2         3         4
1...+....0....+....0....+....0....+....0
```

ファイルの内容:

```
  &NAME1 I= 5, SMITH%P_AGE=40 /
```

Fortran 95

NAMELIST=NEW の指定時に、値のセパレーターのスペースの後ろに **NAMELIST** の注釈が置かれるときの **NAMELIST** の注釈の例を示します。

```
&TODAY I=12345         ! This is a comment. /
X(1)=12345, X(3:4)=2*1.5, I=6,
P="!ISN'T_BOB'S", Z=(123,0)/
```

Fortran 95 の終り

IBM 拡張

NAMELIST=OLD の指定時に、値のセパレーターに続くスペースに **NAMELIST** の注釈が置かれるときの **NAMELIST** 注釈の例を示します。

```
&TODAY I=12345,        ! This is a comment.
X(1)=12345, X(3:4)=2*1.5, I=6,
P="!ISN'T_BOB'S", Z=(123,0) &END
```

IBM 拡張 の終り

名前リスト出力データ

名前リストを使って出力データを書き込むと、名前リストを使って読み取り可能な形式で書き込まれます (ただし、区切られていない文字データを除きます)。名前リストに指定したすべての変数名およびそれらの値が、それぞれのタイプに従って書き込まれます。文字データは **DELIM=** 指定子で指定したように区切られます。データを入れるためのフィールドには、すべての有効数字を収容するために十分な大きさがとられます (フィールドの内容については、 273 ページの表 13 を参照してください)。完全な配列の値は、列順で書き込まれます。

IBM 拡張

名前リストを指定している **WRITE** ステートメントは、名前リスト名を含む単一のレコード、そのレコードに続く、出力データ項目を含む 1 つ以上のレコード、およびスラッシュ (/) 終了マーカーを含む最終レコードの 3 つ以上の出力レコードを作成します。名前リスト出力を受け取るための内部ファイルは、少なくとも 3 つのエレメントを含む文字配列となります。 **WRITE** ステートメント内の転送データの量によって、4 つ以上の配列エレメントが必要となる場合があります。文字変数がデータをすべて保持できる長さであっても、その長い文字変数 1 つだけを使用することはできません。配列エレメントの長さがデータを保持するのに十分でない場合は、4 つ以上の配列エレメントを指定する必要があります。

IBM 拡張 の終り

NAMELIST 実行時オプションが指定されないか、**NAMELIST=NEW** である場合、名前リスト・グループ名および名前リスト項目名は大文字で出力されます。

IBM 拡張

NAMELIST=OLD を指定すると、名前リスト・グループ名および名前リスト項目名が小文字で出力されます。 **-qmixed** コンパイラー・オプションが指定されると、**NAMELIST** 実行時オプションの値に関係なく、名前で大文字小文字の区別がなされます。

NAMELIST=OLD を指定すると、出力レコードの終わりが `&end` によって示されます。

値 **OLD** を指定し、**DELIM=** を指定しないで **NAMELIST** 実行時オプションを指定すると、文字データがアポストロフィで区切られます。区切られていない文字ストリングは、アポストロフィで区切られ、コンマによって分離されます。また、前のレコードから継続している文字ストリングで始まるレコードの先頭にブランクは入れられません。

IBM 拡張 の終り

DELIM= 指定子なしか、または **NONE** の値を持つ **DELIM=** を指定してオープンしたファイル用に出力される文字定数は、次のようになります。
- アポストロフィや引用符によって区切られません。
- 値のセパレーターによって区切られません。
- 1 つのアポストロフィまたは引用符によって外部的に表現される内部的なアポストロフィまたは引用符をそれぞれ持ちます。
- 先行するレコードから継続している文字定数で始まるレコードの先頭で紙送り制御を行うためにプロセッサーによって挿入されたブランク文字を持ちます。

書き込まれた、区切りのない文字データは、文字データとして読み取ることはできません。

IBM 拡張

内部ファイルの場合、文字定数は、**DELIM=** 指定子に、値 **APOSTROPHE** を指定して書き込みます。

IBM 拡張 の終り

値 **QUOTE** の **DELIM=** 指定子付きでオープンしたファイル用に出力された文字定数は、二重引用符によって区切られ、値のセパレーターが前後に入り、2 つの連続する二重引用符によって外部メディア上に表される内部引用符を持ちます。

値 **APOSTROPHE** の **DELIM=** 指定子付きでオープンされたファイル用に出力された文字定数は、アポストロフィによって区切られ、値のセパレーターが前後に入り、2 つの連続するアポストロフィによって外部メディア上に表される内部アポストロフィを持ちます。

IBM 拡張

出力レコードを指定の幅に抑えるには、**OPEN** ステートメント内で **RECL=** 指定子を指定するか、あるいは **NLWIDTH** 実行時オプションを指定します。 **NLWIDTH** 実行時オプションについては、「ユーザーズ・ガイド」を参照してください。

IBM 拡張 の終り

区切られた文字定数が継続する場合を除いて、各出力レコードは、レコードを印刷するときに紙送り制御を行うために、ブランク文字で始まります。

IBM 拡張

外部ファイルの場合は、デフォルトの設定で、出力項目すべてを含むのに十分な大きさの単一の出力レコードにすべての出力項目を指定できます。レコードを別の行に出力するには、(**OPEN** ステートメント内で) **RECL=** 指定子を使用するか、または **NLWIDTH** 実行時オプションを使用します。

IBM 拡張 の終り

非文字の出力データのタイプについては、 272 ページの『リスト指示出力』 を参照してください。

名前リスト出力データの例

```
TYPE PERSON
  INTEGER P_AGE
  CHARACTER(20) P_NAME
END TYPE PERSON
TYPE(PERSON) SMITH
NAMELIST /NL1/ I,J,C,SMITH
CHARACTER(5) :: C='BACON'
INTEGER I,J
I=12046
J=12047
SMITH=PERSON(20,'John Smith')
WRITE(6,NL1)
END
```

NAMELIST=NEW の **WRITE** ステートメントの実行後の出力データ:

```
    1         2         3         4
1...+....0....+....0....+....0....+....0
 &NL1
 I=12046, J=12047, C=BACON, SMITH=20, John Smith
 /
```

IBM 拡張

NAMELIST=OLD の **WRITE** ステートメントの実行後の出力データ:

```
    1         2         3         4
1...+....0....+....0....+....0....+....0
 &nl1
 i=12046, j=12047, c='BACON', smith=20, 'John Smith          '
 &end
```

IBM 拡張 の終り

第 10 章 ステートメントおよび属性

この章では、すべての XL Fortran ステートメントをアルファベット順に説明します。それぞれのステートメントのセクションは、簡単に構文と規則を参照できるように構成されており、『第 1 部 XL Fortran 言語』にあるステートメントの構造および使用法の参照先を示しています。

以下の表では、ステートメントを列挙し、実行可能なステートメント、*specification_part* ステートメント、 **DO** または **DO WHILE** 構造体の終端ステートメントとして使用できるステートメントがそれぞれどれかを示しています。

注:

1. IBM 拡張
2. Fortran 95

表 14. ステートメント表

ステートメント名	実行可能ステートメント	仕様ステートメント	終端ステートメント
ALLOCATABLE		X	
ALLOCATE	X		X
ASSIGN	X		X
AUTOMATIC 1		X	
BACKSPACE	X		X
BLOCK DATA			
BYTE 1		X	
CALL	X		X
CASE	X		
CHARACTER		X	
CLOSE	X		X
COMMON		X	
COMPLEX		X	
CONTAINS			
CONTINUE	X		X
CYCLE	X		
DATA		X	
DEALLOCATE	X		X

表 14. ステートメント表 (続き)

ステートメント名	実行可能ステートメント	仕様ステートメント	終端ステートメント
派生型			
DIMENSION		X	
DO	X		
DO WHILE	X		
DOUBLE COMPLEX 1		X	
DOUBLE PRECISION		X	
ELSE	X		
ELSE IF	X		
ELSEWHERE	X		
END	X		
END BLOCK DATA			
END DO	X		X
END IF	X		
END FORALL 2	X		
END FUNCTION	X		
END INTERFACE		X	
END MAP 1		X	
END MODULE			
END PROGRAM	X		
END SELECT	X		
END SUBROUTINE	X		
END STRUCTURE 1		X	
END TYPE		X	
END UNION 1		X	
END WHERE	X		
ENDFILE	X		X
ENTRY		X	
EQUIVALENCE		X	
EXIT	X		

表 14. ステートメント表 (続き)

ステートメント名	実行可能ステートメント	仕様ステートメント	終端ステートメント
EXTERNAL		X	
FORALL 2	X		X
FORMAT		X	
FUNCTION			
GO TO (割り当て)	X		
GO TO (計算)	X		X
GO TO (無条件)	X		
IF (ブロック)	X		
IF (算術)	X		
IF (論理)	X		X
IMPLICIT		X	
INQUIRE	X		X
INTEGER		X	
INTENT		X	
INTERFACE		X	
INTRINSIC		X	
LOGICAL		X	
MAP 1		X	
MODULE			
MODULE PROCEDURE		X	
NAMELIST		X	
NULLIFY	X		X
OPEN	X		X
OPTIONAL		X	
PARAMETER		X	
PAUSE	X		X
POINTER (Fortran 90)		X	
POINTER (整数) 1		X	
PRINT	X		X

表 14. ステートメント表 (続き)

ステートメント名	実行可能ステートメント	仕様ステートメント	終端ステートメント
PRIVATE		X	
PROGRAM			
PROTECTED 1		X	
PUBLIC		X	
READ	X		X
REAL		X	
RECORD		X	
RETURN	X		
REWIND	X		X
SAVE		X	
SELECT CASE	X		
SEQUENCE		X	
ステートメント関数		X	
STATIC 1		X	
STOP	X		
SUBROUTINE			
STRUCTURE 1		X	
TARGET		X	
TYPE		X	
タイプ宣言		X	
UNION 1		X	
USE		X	
VALUE 1		X	
VIRTUAL 1		X	
VOLATILE 1		X	
WAIT 1	X		X
WHERE	X		X
WRITE	X		X

割り当てステートメントとポインター割り当てステートメントについては、 107 ページの『第 5 章 式および割り当て』で説明します。これらのステートメントはともに実行可能ステートメントであり、終端ステートメントとして働きます。

属性

属性には、それぞれ対応する属性仕様ステートメントが存在します。それぞれの書式を理解するには属性の構文図を参照してください。エンティティーに属性を持たせるには、タイプ宣言ステートメントまたはデフォルト設定を使用します。たとえば、エンティティー *A* に **PRIVATE** 属性を持たせるには、次の方法を使用します。

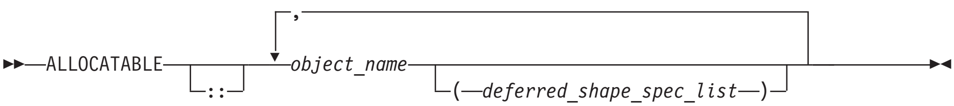
```
REAL, PRIVATE :: A     ! Type declaration statement
PRIVATE :: A           ! Attribute specification statement

MODULE X
  PRIVATE              ! Default setting
  REAL :: A
END MODULE
```

ALLOCATABLE

目的

ALLOCATABLE 属性は、割り振り可能オブジェクト、すなわち **ALLOCATE** ステートメントまたは派生型割り当てステートメントを実行することによって動的にスペースが割り振られるオブジェクトを宣言します。これが配列である場合、据え置き形状配列になります。

構文



object_name 割り振り可能オブジェクトの名前です。

deferred_shape_spec
コロン (:) です。ここで、各コロンは次元を表します。

規則

オブジェクトはポインティング先にはできません。オブジェクトが配列で、有効範囲単位内のどこかに **DIMENSION** 属性で指定された場合、配列指定は `deferred_shape_spec` でなければなりません。

表 15. ALLOCATABLE 属性と互換性のある属性

• AUTOMATIC	• PRIVATE	• STATIC
• DIMENSION	• PROTECTED	• TARGET
• INTENT	• PUBLIC	• VOLATILE
• OPTIONAL	• SAVE	

例

```
REAL, ALLOCATABLE :: A(:,:)  ! Two-dimensional array A declared
                             ! but no space yet allocated
READ (5,*) I,J
ALLOCATE (A(I,J))
END
```

関連情報

- 90 ページの『割り振り可能配列』
- 553 ページの『ALLOCATED(ARRAY) または ALLOCATED(SCALAR)』
- 『ALLOCATE』
- 331 ページの『DEALLOCATE』
- 77 ページの『割り振り状況』
- 90 ページの『据え置き形状配列』
- 203 ページの『仮引き数として割り振り可能なオブジェクト』
- 51 ページの『割り振り可能コンポーネント』

ALLOCATE

目的

ALLOCATE ステートメントは、ポインター・ターゲットと割り振り可能オブジェクトにストレージを動的に提供します。

構文

```
>>—ALLOCATE—(—allocation_list——————————————————)——————————><
                               └,—STAT— = —stat_variable┘
```

stat_variable
スカラー整変数です。

allocation_list

```
►►─allocate_object─┬──────────────────────────────────────────┬──────────►◄
                   │    ┌─,───────────────────────────┐       │
                   │    ▼                             │       │
                   └─(──┬──────────────────┬─upper_bound─┴─)──┘
                        └─lower_bound──:─┘
```

allocate_object
: 変数名あるいは構造体コンポーネントです。これはポインターまたは割り振り可能オブジェクトでなければなりません。

lower_bound、*upper_bound*
: スカラー整数式です。

規則

ポインターに対して **ALLOCATE** ステートメントを実行すると、ポインターは、割り振られたターゲットと関連させられます。割り振り可能オブジェクトに対して実行すると、オブジェクトは定義可能になります。

指定する次元の数 (つまり *allocation* 内の上限の境界の数) は、*allocate_object* のランクと等しくなければなりません。 **ALLOCATE** ステートメントが配列に対して実行された場合、境界の値はその時点で決定されます。境界式の中でそれに続くエンティティーの再定義や未定義は配列指定には影響しません。下限値が省略された場合は、デフォルトの値 1 が割り当てられます。下限値が上限値を超えた場合、次元は 0 になり、*allocate_object* のサイズも 0 になります。

allocate_object、または *allocate_object* の指定された境界は、 *stat_variable*、または同じ ALLOCATE ステートメント内にある *allocate_object* の値、境界、割り振り状況、または関連付け状況に依存しません。

stat_variable は、それが現れる ALLOCATE ステートメントの中に割り振られません。また、同じ ALLOCATE ステートメント内にある *allocate_object* の値、境界、割り振り状況、または関連付け状況に依存しません。

STAT= 指定子が存在せず、このステートメントの実行中にエラーが発生した場合、プログラムは終了します。 **STAT=** 指定子が存在する場合、*stat_variable* には、以下の値の 1 つが割り当てられます。

IBM 拡張

Stat 値	エラー状態
0	エラーなし
1	割り振りを試みているシステム・ルーチンにエラー
2	割り振りに無効なデータ・オブジェクトが指定された

Stat 値	エラー状態
3	1 と 2 の両方のエラーが発生した

IBM 拡張 の終り

すでに割り振り済みの割り振り可能オブジェクトを割り振ると、 **ALLOCATE** ステートメントでエラー状態が発生します。

ポインター割り振りは、**TARGET** 属性を持つオブジェクトを作成します。ポインター指定を実行することによって、追加のポインターをこのターゲット (またはターゲットのサブオブジェクト) に関連付けることができます。すでにターゲットに関連したポインターを再度割り振ると、以下のようになります。
- 新しいターゲットが作成され、ポインターはこのターゲットと関連付けられます。
- そのポインターとの以前の関連付けがすべて破壊されます。
- 以前に割り振りによって作成され、現在他のどのポインターとも関連付けられていないすべてのターゲットはアクセス不能になります。

派生型のオブジェクトが ALLOCATE ステートメントによって作成されると、割り振り可能最終コンポーネントの割り振り状況は、現在、割り振りが行われていないという状況になります。

割り振り可能オブジェクトが現在割り振られているかどうかを判別するには、**ALLOCATED** 組み込み関数を使用します。ポインターの関連付け状況、およびポインターが現在指定しているターゲットに関連付けられているかどうかを調べるには、**ASSOCIATED** 組み込み関数を使用します。

例

```
CHARACTER, POINTER :: P(:,:)
CHARACTER, TARGET :: C(4,4)
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
P => C
N = 2; M = N
ALLOCATE (P(N,M),STAT=I)         ! P is no longer associated with C
N = 3                            ! Target array for P maintains 2X2 shape
IF (.NOT.ALLOCATED(A)) ALLOCATE (A(N**2))
END
```

関連情報

- 289 ページの『ALLOCATABLE』
- 331 ページの『DEALLOCATE』
- 77 ページの『割り振り状況』
- 167 ページの『ポインター関連付け』
- 90 ページの『据え置き形状配列』

- 553 ページの『ALLOCATED(ARRAY) または ALLOCATED(SCALAR)』
- 557 ページの『ASSOCIATED (POINTER, TARGET)』
- 6 ページの『初期化』
- 203 ページの『仮引き数として割り振り可能なオブジェクト』
- 51 ページの『割り振り可能コンポーネント』

ASSIGN

目的

ASSIGN ステートメントはステートメント・ラベルを整変数に割り当てます。

構文

```
▶▶──ASSIGN──stmt_label──TO──variable_name──────────────────────────────▶◀
```

stmt_label
: **ASSIGN** ステートメント・ラベルを含む有効範囲単位の、実行可能ステートメントまたは **FORMAT** ステートメントのステートメント・ラベルを指定します。

variable_name
: スカラー **INTEGER(4)** または **INTEGER(8)** 変数の名前です。

規則

指定されたステートメント・ラベルを含むステートメントは **ASSIGN** ステートメントと同じ有効範囲単位に存在しなければなりません。

- ステートメント・ラベルを含むステートメントが実行可能ステートメントである場合は、そのラベル名を、同じ有効範囲単位内の割り当て済みの **GO TO** ステートメントの中で使用できます。
- ステートメント・ラベルを含むステートメントが **FORMAT** ステートメントである場合は、そのラベル名を、同じ有効範囲単位内の **READ**、**WRITE**、または **PRINT** ステートメントの中で形式指定子として使用できます。

ラベルの値で定義した整変数を、同じ、または異なるラベル、または整数値で再定義できます。ただし、割り当て済みの **GO TO** ステートメントの中で、または I/O ステートメントの中の形式識別子として変数を参照する前に、ラベル値を使用して変数を定義しなければなりません。

variable_name の値はラベルによって示される整定数ではないので、その値をラベルとして使用することはできません。

Fortran 95

ASSIGN ステートメントは Fortran 95 からは削除されています。

Fortran 95 の終り

例

```
      ASSIGN 30 TO LABEL
      NUM = 40
      GO TO LABEL
      NUM = 50              ! This statement is not executed
30    ASSIGN 1000 TO IFMT
      PRINT IFMT, NUM       ! IFMT is the format specifier
1000  FORMAT(1X,I4)
      END
```

関連情報

- 14 ページの『ステートメント・ラベル』
- 382 ページの『GO TO (割り当て)』
- 927 ページの『削除された機能』

AUTOMATIC

IBM 拡張

目的

AUTOMATIC 属性は変数に自動ストレージ・クラスを持たせることを指定します。つまり、一度プロシージャーが終了した後は変数は定義済みではありません。

構文

```
>>—AUTOMATIC—┬────┬—automatic_list——————————————————————————————><
             └—::—┘
```

automatic
: 変数名、または明示的形状指定リストか据え置き形状指定リストを持つ配列宣言子です。

規則

automatic の名前がそれが宣言されている関数と同じ名前の場合、文字タイプ、あるいは派生型にすることはできません。

ポインターまたは配列である関数結果、仮引き数、ステートメント関数、自動オブジェクト、またはポインティング先には、**AUTOMATIC** 属性を指定してはなりません。**AUTOMATIC** 属性を持つ変数をモジュールの有効範囲単位内で定義することはできません。共通ブロック項目に、**AUTOMATIC** 属性によって明示的に宣言された変数を指定することはできません。

同じ有効範囲単位内で 1 つの変数に対して複数の **AUTOMATIC** 属性を指定することはできません。

スレッドの作業範囲内で **AUTOMATIC** と宣言された変数はすべて、そのスレッドに対してローカルになります。

-qinitauto コンパイラー・オプションが指定されていない場合、**DATA** ステートメントやタイプ宣言ステートメントを使用しても **AUTOMATIC** 属性を持つ変数を初期化することはできません。 **-qinitauto** オプションが指定されている場合、**AUTOMATIC** 属性を持つ変数のストレージのすべてのバイトは指定されたバイト値 (値が設定されていない場合は 0) に初期化されます。

automatic がポインターの場合、**AUTOMATIC** 属性はポインター自身に設定されるのであり、そのポインターに関連する (あるいは関連する可能性のある) ターゲットに設定されることはありません。

ローカル変数はデフォルトで自動ストレージ・クラスを持っています。呼び出しコマンドに関するデフォルト設定値の詳細については、「ユーザーズ・ガイド」の『**-qsave** オプション』を参照してください。

注: **AUTOMATIC** 属性を持つオブジェクトを自動オブジェクトと混同しないでください。 29 ページの『自動オブジェクト』を参照してください。

AUTOMATIC 属性と互換性のある属性

- ALLOCATABLE
- DIMENSION
- POINTER
- TARGET
- VOLATILE

例

```
CALL SUB
CONTAINS
  SUBROUTINE SUB
    INTEGER, AUTOMATIC :: VAR
    VAR = 12
  END SUBROUTINE                  ! VAR becomes undefined
END
```

関連情報

- 78 ページの『変数のストレージ・クラス』
- 「ユーザーズ・ガイド」の『**-qinitauto** オプション』

IBM 拡張 の終り

BACKSPACE

目的

BACKSPACE ステートメントは、順次アクセス、または定様式ストリーム・アクセス用に接続された外部ファイルを位置付けます。

構文

```
►►──BACKSPACE──┬─u───────────────────┬──────────────────────────────►◄
               └─(──position_list──)─┘
```

u 外部装置識別子です。 *u* の値はアスタリスク、またはホレリス定数であってはなりません。

position_list
: 装置指定子 ([**UNIT=**]*u*) を必ず 1 つ含んでいなければならないリストです。このリストには、他の有効な各指定子を 1 つずつ入れることができます。

[UNIT=] *u*
: 装置指定子です。*u* は外部装置識別子で、その値はアスタリスクであってはなりません。外部装置識別子はスカラー整数式 (1 ～ 2147483647 の値を持つ) で表される外部ファイルを示します。オプションの文字である **UNIT=** を省略する場合は、*u* は *position_list* の最初の項目でなければなりません。

IOSTAT= *ios*
: I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。 **BACKSPACE** ステートメントの実行が完了すると、*ios* の値は次のように定義されます。

  - エラーが発生しなかった場合は 0
  - エラーが発生した場合は正の値

ERR= *stmt_label*
: エラーが発生した場合に制御が移される、同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子は、エラー・メッセージを抑制します。

規則

BACKSPACE ステートメントを実行すると、現行レコードがある場合は、ファイルは現行レコードの前に位置付けられます。現行レコードがない場合には、ファイルは前のレコードの前に位置付けられます。ファイルが初期点にある場合には、ファイル位置は変更されません。

リスト指示形式設定または名前リスト形式設定によって書き込んだレコードに BACKSPACE を使用することはできません。

順次アクセスで、先行レコードがファイル終了レコードである場合、ファイル終了レコードの前にファイルが位置付けられます。

ERR= と **IOSTAT=** 指定子が設定されているときにエラーが検出されると、**ERR=** 指定子によって指定されたステートメントに対して転送が行われ、正の整数値が *ios* に割り当てられます。

IBM 拡張

IOSTAT= も **ERR=** も指定していない場合は、以下のとおりです。

- 重大なエラーが検出されるとプログラムは停止します。
- 回復可能エラーが検出され、 **ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、プログラムは次のステートメントの処理を継続します。オプションが **NO** に設定されていると、プログラムは停止します。

IBM 拡張 の終り

例

```
   BACKSPACE 15
   BACKSPACE (UNIT=15,ERR=99)
   ⋮
99 PRINT *, "Unable to backspace file."
   END
```

関連情報

- 229 ページの『条件および IOSTAT 値』
- 217 ページの『第 8 章 I/O の概念』
- 「ユーザーズ・ガイド」の『実行時オプションの設定』

BLOCK DATA

目的

BLOCK DATA ステートメントはブロック・データ・プログラム単位の最初のステートメントであり、名前付き共通ブロック内の変数に初期値を与えます。

構文

```
►►──BLOCK DATA──┬─────────────────┬──────────────────────────────────────────►◄
                └─block_data_name─┘
```

block_data_name
ブロック・データ・プログラム単位の名前です。

規則

1 つの実行可能プログラム内に複数のブロック・データ・プログラム単位を入れることができますが、名前を指定しなくてよいのはその中の 1 つだけです。

ブロック・データ・プログラム単位の名前を指定する場合、その名前は実行可能プログラム内のサブプログラム、エントリー、メインプログラム、モジュール、または共通ブロックの名前と同じであってはなりません。また、当該プログラム単位内のローカル・エンティティーの名前と同じであってもなりません。

例

```
BLOCK DATA ABC
  PARAMETER (I=10)
  DIMENSION Y(5)
  COMMON /L4/ Y
  DATA Y /5*I/
END BLOCK DATA ABC
```

関連情報

- 187 ページの『ブロック・データのプログラム単位』
- **END BLOCK DATA** ステートメントの詳細については、 350 ページの『END』

BYTE

IBM 拡張

目的

BYTE タイプ宣言ステートメントはオブジェクトとバイト・タイプの関数の属性を指定します。各スカラー・オブジェクトの長さは 1 です。オブジェクトには初期値を割り当てることができます。

構文

```
▸▸─BYTE─┬─::──────────────────┬─entity_decl_list──────────────▸◂
         └─,─attr_spec_list─::─┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

attr_spec
: 特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
: **IN**、**OUT**、または **INOUT** のいずれかです。

::
: ダブル・コロン・セパレーターです。複数の属性を指定するとき、または = *initialization_expr* あるいは => **NULL()** を使用するときに必要になります。

array_spec
次元境界のリストです。

entity_decl

```
►►─a─┬──────────────────┬─┬─────────────────────────────┬──────────────────────►◄
     └─(─array_spec─)─┘ ├─/─initial_value_list─/─┤
                          ├─=─initialization_expr──┤
                          └─ => ─NULL()────────────┘
```

a オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

=> NULL()
ポインター・オブジェクトに初期値を与えます。

規則

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合は、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。ブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、またはモジュール内の名前付き共通ブロックにある場合、そのオブジェクトは初期化することができます。

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。*initialization_expr* F95 または **NULL()** F95 が指定されていて、

- エンティティーが変数の場合、変数が最初に定義されます。

Fortran 95

- エンティティーが派生型コンポーネントの場合、派生型にはデフォルトの初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。

変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、*a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

事前に定数の名前として定義されている `T` または `F` がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

例

```
BYTE, DIMENSION(4) :: X=(/1,2,3,4/)
```

関連情報

- 41 ページの『バイト』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

IBM 拡張 の終り

CALL

目的

CALL ステートメントは実行するサブルーチンを呼び出します。

構文

```
>>──CALL──name──┬─────────────────────────────────────┬───────────────><
                └─(──┬──────────────────────────┬──)─┘
                     └─actual_argument_spec_list─┘
```

name 内部、外部、またはモジュール・サブルーチン、外部またはモジュール・サブルーチンのエントリー、組み込みサブルーチンの名前、または総称名です。

規則

CALL ステートメントを実行すると次のイベントが順番に発生します。

1. 式である実引き数が評価されます。
2. 実引き数が、対応する仮引き数に関連付けられます。

3. 指定されたサブルーチンに制御が移ります。
4. サブルーチンが実行されます。
5. サブルーチンから制御が戻ります。

サブルーチン・ステートメントで **RECURSIVE** キーワードを指定していると、サブプログラムはそれ自身を再帰的、直接的、あるいは間接的に呼び出すことができます。

IBM 拡張

-qrecur コンパイラー・オプションを指定した場合、外部サブプログラムもそれ自身を直接的、あるいは間接的に参照することができます。

IBM 拡張 の終り

CALL ステートメントの引き数の中に 1 つ以上の選択戻り指定がある場合、**RETURN** ステートメントのサブルーチンによって指定されたアクションによって、指定したステートメント・ラベルの 1 つに制御を移すことができます。

IBM 拡張

値または参照により引き数を引き渡すことによって言語間呼び出しを容易にするために、それぞれの場合に使用するための引き数リスト組み込み関数 **%VAL** および **%REF** が提供されています。それらは、Fortran 以外のプロシージャー参照にのみ使用することができます。

IBM 拡張 の終り

値による引き数の引き渡しには、**VALUE** 属性を使用することもできます。

例

```
INTERFACE
  SUBROUTINE SUB3(D1,D2)
    REAL D1,D2
  END SUBROUTINE
END INTERFACE
ARG1=7 ; ARG2=8
CALL SUB3(D2=ARG2,D1=ARG1)    ! subroutine call with argument keywords
END

SUBROUTINE SUB3(F1,F2)
  REAL F1,F2,F3,F4
  F3 = F1/F2
  F4 = F1-F2
  PRINT *, F3, F4
END SUBROUTINE
```

関連情報

- 209 ページの『再帰』
- 197 ページの『%VAL および %REF』
- 193 ページの『実引き数の仕様』
- 206 ページの『仮引き数としてのアスタリスク』

CASE

目的

CASE ステートメントは、**CASE** 構造体の中の **CASE** ステートメント・ブロックを初期化します。また、このステートメントは 1 つのブロックを実行のために選択するための、非常に簡潔な構文を備えています。

構文

```
>>─CASE─case_selector─┬─────────────────────┬────────────────────────><
                      └─case_construct_name─┘
```

case_selector

```
>>─┬─DEFAULT──────────────────────────────────────────┬──────────────><
   │   ┌─,─────────────────────────────────────┐      │
   │   ▼                                       │      │
   └─(───┬─case_value──────────────────────────┼──)──┘
         ├─low_case_value─:─high_case_value─┤
         ├─low_case_value─:─────────────────┤
         └─:─high_case_value────────────────┘
```

case_construct_name
: **CASE** 構造体を識別する名前です。

case_value
: 整数、文字、または論理タイプのスカラー初期化式です。

low_case_value、*high_case_value*
: これらは、それぞれ整数、文字、または論理タイプのスカラー初期化式です。

規則

SELECT CASE ステートメントにより決定するケース指標は、**CASE** ステートメント内の各 *case_selector* と比較されます。一致すると、**CASE** ステートメントに関連する *stmt_block* が実行されます。一致しないと、*stmt_block* は実行されません。どのケース値範囲もオーバーラップすることはできません。

一致があるかどうかは次のように判別します。

case_value

データ型:　整数、文字、または論理
整数および文字の一致:　*case index* = *case_value*
論理の一致:　*case index* **.EQV.** *case_value* が真

low_case_value : high_case_value

データ型:　整数または文字
一致:　*low_case_value* ≤ *case index* ≤ *high_case_value*

low_case_value :

データ型:　整数または文字
一致:　*low_case_value* ≤ *case index*

: high_case_value

データ型:　整数または文字
一致:　*case index* ≤ *high_case_value*

DEFAULT

データ型:　適用外
一致:　他に一致がない場合

複数の一致があってはなりません。一致が 1 つの場合、その一致した *case_selector* に関連するステートメント・ブロックが実行され、ケース構造体の実行が完了します。一致がない場合、ケース構造体の実行が完了します。

case_construct_name を指定する場合は、**SELECT CASE** ステートメントおよび **END SELECT** ステートメントに指定した名前に一致していなければなりません。

DEFAULT はデフォルトの *case_selector* です。 **CASE** ステートメントのうち *case_selector* として **DEFAULT** を持てるのは 1 つだけです。

各ケース値は、**SELECT CASE** ステートメントに定義してある *case_expr* と同じデータ型でなければなりません。タイプなし定数または **BYTE** 名前付き定数が *case_selectors* で検出された場合、その定数は *case_expr* のデータ型に変換されます。

case_expr およびケース値が文字タイプの場合、それらの長さは異なってもかまいません。 **-qctyplss** コンパイラー・オプションを指定すると、*case_expr* として使用される文字定数が文字タイプとして残ります。文字定数式はタイプなし定数としては処理されません。

例

```
ZERO: SELECT CASE(N)

  CASE DEFAULT ZERO          ! Default CASE statement for
                             ! CASE construct ZERO
       OTHER: SELECT CASE(N)
          CASE(:-1)          ! CASE statement for CASE
                             ! construct OTHER
             SIGNUM = -1
          CASE(1:) OTHER
              SIGNUM = 1
       END SELECT OTHER
  CASE (0)
    SIGNUM = 0

END SELECT ZERO
```

関連情報

- 149 ページの『CASE 構造体』
- 465 ページの『SELECT CASE』
- **END SELECT** ステートメントの詳細については、351 ページの『END (構造体)』

CHARACTER

目的

CHARACTER タイプ宣言ステートメントは文字タイプのオブジェクトと関数の種類、長さ、および属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>-CHARACTER--+---------------+--+----------------------+--entity_decl_list-><
              '-char_selector-'  '-,--attr_spec_list--::-'
                                 ::
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

char_selector

文字長を指定します。

IBM 拡張

これは 0 から 256 MB までの文字数です。 256 MB を超える値は 256 MB に設定されます。負の値はゼロに設定されます。値を指定しない場合、デフォルトの長さは 1 です。kind 型付きパラメーターを指定した場合、値は 1 でなければなりません。その値は ASCII 文字を表します。

IBM 拡張 の終り

```
►►──┬─(─┬─LEN─=─type_param_value─,─KIND─=─int_init_expr──────────┬─)─┬──────────────────►◄
    │   ├─type_param_value─,─┬──────────┬─int_init_expr───────────┤   │
    │   │                    └─KIND─=─┘                          │   │
    │   ├─KIND─=─int_init_expr─┬────────────────────────────┬─────┤   │
    │   │                      └─,─LEN─=─type_param_value─┘     │   │
    │   └─┬─────────┬─type_param_value────────────────────────────┘   │
    │     └─LEN─=─┘                                                   │
    └─*─char_length─┬───┬──────────────────────────────────────────────┘
                    └─,─┘
```

type_param_value
: 宣言式またはアスタリスク (*) です。

int_init_expr
: スカラー整数初期化式です。この式の値は 1 でなければなりません。

char_length
: 括弧内の *type_param_value* か、またはスカラー整数リテラル定数 (この定数は kind 型付きパラメーターを指定することはできません) です。

attr_spec
: 特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
: **IN**、**OUT**、または **INOUT** のいずれかです。

::
: ダブル・コロン・セパレーターです。複数の属性を指定する場合、あるいは = *initialization_expr* F95 または => **NULL()** F95 を使用する場合に必要になります。

array_spec
: 次元境界のリストです。

entity_decl

```
►►──a──┬──────────────────────────────────────────────┬──────────────────►
       │ └─ * ──char_length─┘  └─(──array_spec──)─┘  │
       └─(──array_spec──)── * ──char_length───────────┘

►──┬──────────────────────────────────────┬───────────────────────────────►◄
   │                          (1)         │
   └─┬─/──initial_value_list──/─────────┬─┘
     ├─ = ──initialization_expr─────────┤
     │              (2)                 │
     └─ => ──NULL()─────────────────────┘
```

注:

1 IBM 拡張

2 Fortran 95

a オブジェクト名または関数名です。 *array_spec* を関数名に指定するこ

とはできません。

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。以下の場合にオブジェクトを初期化できます。

- オブジェクトが、ブロック・データ・プログラム単位内の名前付き共通ブロックにある。

IBM 拡張

- オブジェクトがモジュール内の名前付き共通ブロックにある。

IBM 拡張 の終り

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* または *type_param_value* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。*initialization_expr* F95 または **NULL()** F95 が指定されていて、

- エンティティーが変数の場合、変数が最初に定義されます。

Fortran 95

- エンティティーが派生型コンポーネントの場合、派生型にはデフォルトの初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。

変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、*a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は、**DIMENSION** の中で指定されている *array_spec* に優先します。 *entity_decl* の中で指定されている *char_length* は、*char_selector* で指定されているどの長さよりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている `T` または `F` がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

ステートメント内にダブル・コロン・セパレーター (::) がない場合にのみ、**CHARACTER** タイプ宣言ステートメントの *char_length* の後にオプションのコンマを入れることができます。

CHARACTER タイプ宣言ステートメントがモジュール、ブロック・データ・プログラム単位、またはメインプログラムの有効範囲にあり、エンティティーの長さを継承されるものとして指定する場合、そのエンティティーは名前付き文字定数の名前でなければなりません。文字定数は **PARAMETER** 属性に定義される対応する式の長さになります。

CHARACTER タイプ宣言ステートメントがプロシージャーの有効範囲にあり、エンティティーの長さが継承される場合、そのエンティティーの名前は仮引き数または名前付

き文字定数の名前でなければなりません。ステートメントが外部関数の有効範囲にある場合、そのステートメントを同じプログラム単位内の **FUNCTION** または **ENTRY** ステートメント内の関数名またはエントリー名にすることができます。エンティティー名が仮引き数の名前の場合、仮引き数はプロシージャーを参照するために、関連する実際の引き数の名前を受け入れます。エンティティー名が文字定数の名前と同じ場合、文字定数は **PARAMETER** 属性が定義する対応した式の長さを受け入れます。エンティティー名が関数名またはエントリー名と同じ場合、エンティティーは呼び出し側の有効範囲単位内で指定されている長さを受け入れます。

文字関数の長さは、関数のタイプがインターフェース・ブロックで宣言されていない場合は定数式でなければならない宣言式になり、仮プロシージャー名の長さを示す場合はアスタリスクになります。内部関数、モジュール関数、または再帰的関数の場合、または関数が配列またはポインティング値を返す場合、長さにアスタリスクを使うことはできません。

例

```
I=7
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER(7), TARGET :: ORANGES = 'ORANGES'
CALL TEST(APPLES,I)
CONTAINS
  SUBROUTINE  TEST(VARBL,I)
    CHARACTER*(*), OPTIONAL :: VARBL   ! VARBL inherits a length of 6
    CHARACTER(I) :: RUNTIME            ! Automatic object with length of 7
  END SUBROUTINE
END
```

関連情報

- 37 ページの『文字』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』
- 「ユーザーズ・ガイド」の『**-qcharlen** オプション』を参照してください。

CLOSE

目的

CLOSE ステートメントは外部ファイルを装置から切断します。

構文

```
▶▶—CLOSE—(—close_list—)————————————————————————————————▶◀
```

close_list

装置指定子 (**UNIT=***u*) を必ず 1 つ含んでいなければならないリストです。このリストには、許可されている他の指定子をそれぞれ 1 つずつ入れることができます。有効な指定子は次のとおりです。

[UNIT=] *u*

装置指定子です。*u* は外部装置識別子で、その値はアスタリスクであってはなりません。外部装置識別子はスカラー整数式 (1 ～ 2147483647 の値を持つ) で表される外部ファイルを示します。オプションの文字である **UNIT=** を省略する場合は、*u* は *close_list* の最初の項目でなければなりません。

IOSTAT= *ios*

I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。このステートメントの実行が完了すると、*ios* は次の値に定義されます。

- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値

ERR= *stmt_label*

エラーが発生した場合に制御が移される、同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。**ERR=** 指定子は、エラー・メッセージを抑制します。

STATUS= *char_expr*

ファイルのクローズ後の状況を示します。 *char_expr* はスカラー文字式で、式の値は、後続ブランクを除去すると **KEEP** または **DELETE** のいずれかになります。

- 既存のファイルに対して **KEEP** を指定した場合、 **CLOSE** ステートメントを実行した後も、そのファイルは存在し続けます。存在しないファイルに対して **KEEP** を指定した場合、 **CLOSE** ステートメントを実行した後にも、そのファイルは存在しません。**CLOSE** ステートメントの実行前の状況が **SCRATCH** であるファイルに対して **KEEP** を指定することはできません。
- **DELETE** を指定した場合、**CLOSE** ステートメントを実行すると、そのファイルは存在しなくなります。

ファイル状況が **SCRATCH** の場合、デフォルトは **DELETE** です。それ以外の場合、デフォルトは **KEEP** です。

規則

装置を参照する **CLOSE** ステートメントは実行可能プログラムのどのプログラム単位の中に置いてもかまいません。その装置を参照する **OPEN** ステートメントと同じ有効範囲単位内に置く必要はありません。存在しない装置またはファイルが接続されていない装置を指定することもできますが、その場合、**CLOSE** ステートメントは何の効力も持ちません。

▶ IBM 装置 0 はクローズできません。 IBM ◀

実行可能プログラムが、エラー以外の理由によって停止すると、接続されているすべての装置はクローズされます。終了前のファイル状況が **SCRATCH** でない限り、装置は **KEEP** 状態でクローズされます。**SCRATCH** の場合、装置は **DELETE** 状態でクローズされます。結果として、**STATUS=** 指定子の付いていない **CLOSE** ステートメントが、その接続装置で実行されたかのようになります。

事前に接続されていた装置が **CLOSE** ステートメントによって切断された場合、その装置が後で **WRITE** ステートメントの中で指定されると (明示的にオープンされずに)、暗黙的なオープン規則が適用されます。

例

```
CLOSE(15)
CLOSE(UNIT=16,STATUS='DELETE')
```

関連情報

- 222 ページの『装置の接続』
- 229 ページの『条件および IOSTAT 値』
- 420 ページの『OPEN』

COMMON

目的

COMMON ステートメントは共通ブロックとその内容を指定します。共通ブロックとは複数の有効範囲単位が共有できる 1 つのストレージのことです。共通ブロックを使うことによって、複数のプログラム単位で同一のデータを定義して参照し、ストレージを共有することが可能になります。

構文

```
►►──COMMON──┬─────────────────────────────┬──object_list──────────────────►
            └─/──┬───────────────────┬─/─┘
                 └─common_block_name─┘

►─┬──────────────────────────────────────────────────────────────┬───────►◄
  │ ┌──────────────────────────────────────────────────────────┐ │
  │ ▼                                                          │ │
  └───┬───┬─/──┬───────────────────┬─/──object_list────────────┴─┘
      └─,─┘    └─common_block_name─┘
```

object

```
►►──variable_name──┬──────────────────────────────────────┬──────────────►◄
                   └─(──explicit_shape_spec_list──)──────┘
```

規則

object は、仮引き数、自動オブジェクト、割り振り可能オブジェクト、割り振り可能最終コンポーネントを持つ派生型のオブジェクト、ポインティング先、関数、関数結果、またはプロシージャーへのエントリーを参照することはできません。 *object* は **STATIC** または **AUTOMATIC** 属性を持つことはできません。

explicit_shape_spec_list がある場合、*variable_name* に **POINTER** 属性を持たせることはできません。各次元境界は定数宣言式でなければなりません。この書式は *variable_name* が **DIMENSION** 属性を持つことを指定します。

派生型の *object* の場合は、順序派生型でなければなりません。すべての最終コンポーネントが非ポインターであり、またすべてが非文字タイプ (または倍精度実数) であるか、あるいは文字タイプのいずれかである順序構造体の場合、構造体は、そのコンポーネントが共通ブロック内で直接列挙されているかのように処理されます。

共通ブロック内のポインター・オブジェクトとなれるのは、同じタイプ、型付きパラメーター、およびランクのポインターに関連したストレージだけです。

TARGET 属性を持つ共通ブロック内のオブジェクトは、別のオブジェクトとストレージを関連させることができます。オブジェクトは **TARGET** 属性を持ち、タイプと型付きパラメーターが同じでなければなりません。

IBM 拡張

BYTE タイプのポインターを **INTEGER(1)** および **LOGICAL(1)** タイプのポインターに関連するストレージにすることができます。 **-qintlog** コンパイラー・オプションを指

定すると、同じ長さの整数および論理ポインターは関連したストレージになります。

IBM 拡張 の終り

common_block_name を指定すると、*object_list* で指定されたすべての変数はその名前付き共通域ブロック内にあることを宣言されます。 *common_block_name* を省略すると、*object_list* で指定したすべての変数は無名共通ブロック内に置かれます。

1 つの有効範囲単位内では、同じ共通ブロックを同じ **COMMON** ステートメントあるいは別の **COMMON** ステートメントに複数回指定してもかまいません。同じ共通ブロックを指定するたびに、その名前で指定した共通ブロックが参照されます。共通ブロック名はグローバル・エンティティーです。

共通ブロック内の変数のデータ型は異なっていてもかまいません。文字データ型と非文字データ型を 1 つの共通ブロック内に混在させることができます。共通ブロック内の変数名を指定できるのは 1 つの有効範囲単位内では 1 つの **COMMON** ステートメントだけです。また、共通ブロック内の変数名を同一の **COMMON** ステートメント内で重複させることはできません。

IBM 拡張

デフォルトでは、共通ブロックはスレッドをまたがって共用されます。そのため、共通ブロック内のストレージ単位が、1 つ以上のスレッドによって更新される必要がある場合や、1 つのスレッドで更新され、別のスレッドから参照される場合に、**COMMON** ステートメントを使用すると、スレッドの安全性は守られなくなります。アプリケーションで、スレッドの安全性を確保したうえで **COMMON** を使用するには、ロックを使ってデータへのアクセスを逐次化するか、または、確実に共通ブロックをおのおののスレッドに対してローカルにしてください。 **Pthreads** ライブラリー・モジュールには、ロックを使ってデータへのアクセスを逐次化するための mutex が備わっています。詳細については、787 ページの『第 15 章 Pthreads ライブラリー・モジュール』を参照してください。また、**CRITICAL** ディレクティブの *lock_name* 属性にも、データへのアクセスを逐次化するための機能が備わっています。 詳細については、701 ページの『CRITICAL / END CRITICAL』を参照してください。 **THREADLOCAL** および **THREADPRIVATE** ディレクティブを使用すると、共通ブロックを確実にそれぞれのスレッドのローカルにすることができます。詳細については、738 ページの『THREADLOCAL』および 741 ページの『THREADPRIVATE』を参照してください。

IBM 拡張 の終り

共通関連付け

1 つの実行可能プログラム域では、サイズが 0 でない同じ名前のすべての名前付き共通ブロックは始点を同じくする同じ記憶単位を持ちます。 1 つの実行可能プログラム内に

は、無名共通ブロックを 1 つ置くことができます。サイズが 0 ではない無名共通ブロックを参照するすべての有効範囲単位は、始点を同じくする同じ有効範囲単位を参照します。

サイズが 0 で名前が同じすべての共通ブロックは、同じストレージを共用します。サイズが 0 であるすべての無名共通ブロックは、サイズが 0 でない無名共通ブロックの最初の記憶単位と同じストレージを共用します。使用関連付けまたはホスト関連付けを使用することにより、同一の有効範囲単位内でこれらの関連オブジェクトにアクセスできるようになります。

関連付けは記憶単位ごとに実行されるため、共通ブロック内の変数の名前およびタイプは別の有効範囲単位では異なっていてもかまいません。

共通ブロックのストレージの順序: 有効範囲単位の共通ブロック内の変数には、**COMMON** ステートメントで名前が現れる順に、ストレージが割り当てられます。

EQUIVALENCE ステートメントを使用して共通ブロックを拡張できますが、拡張部分は最後のエントリーの後に追加できるだけで、最初のエントリーの前には追加できません。たとえば、次のステートメントでは X を指定します。

```
COMMON /X/ A,B      ! common block named X
REAL C(2)
EQUIVALENCE (B,C)
```

共通ブロック X の内容は、次のように指定されます。

```
|    |    |    |    |    |    |    |
|    |    |    |    |
Variable A: |         A         |
Variable B:                     |         B         |
Array C:                        |        C(1)       |        C(2)
|
```

有効範囲単位内の共通ブロックのストレージの順番に影響を与えるステートメントは **COMMON** と **EQUIVALENCE** です。使用関連付けまたはホスト関連付けによって共通にアクセス可能になる変数はこの限りではありません。

EQUIVALENCE ステートメントを使って、2 つの異なる共通ブロックの記憶順序を関連させることはできません。モジュールの有効範囲単位内で共通ブロックを宣言することはできますが、使用関連付けを介してモジュールからエンティティーにアクセスする他の有効範囲単位内で共通ブロックを宣言することはできません。

COMMON を使用するとデータの境界合わせが正しく実行されない場合があります。境界合わせが正しく実行されなかったデータを使用すると、プログラムのパフォーマンスに悪影響を与えます。

共通ブロックのサイズ: 共通ブロックのサイズはストレージのバイト数に等しくなります。それは共通ブロック内のすべての変数を収容するのに必要なサイズです。これには、等価関連付けによって拡張された部分も含まれます。

名前付き共通ブロックと無名共通ブロックの相違点:

- 1 つの実行可能プログラム内に、名前付き共通域は複数あってもかまいませんが、無名共通ブロックは 1 つに限られます。
- 1 つの実行可能プログラムの有効範囲単位内で、同じ名前の複数の共通ブロックは同じサイズでなければなりませんが、無名共通ブロックのサイズは互いに異なっていてもかまいません。 (複数の有効範囲単位内でサイズの異なる複数の無名共通ブロックを指定した場合、そのうちで最長のブロックの長さが実行可能プログラム内の無名共通ブロックの長さになります。)
- 名前付き共通域ブロック内のオブジェクトを最初に定義するために、 **DATA** ステートメントまたはタイプ宣言ステートメントを含む **BLOCK DATA** プログラム単位を使用することができます。無名共通ブロック内の共通ブロックのエレメントを最初に定義することはできません。

  名前付き共通ブロックまたはその一部を複数の有効範囲単位内で初期化すると、初期値は未定義になります。このような問題を回避するには、ブロック・データ・プログラム単位 IBM またはモジュール IBM を使用して名前付き共通ブロックを初期化してください。初期化は、各名前付き共通ブロックごとに、それぞれ 1 つのブロック・データ・プログラム単位 IBM またはモジュール IBM 内だけで行う必要があります。

例

```
INTEGER MONTH,DAY,YEAR
COMMON /DATE/ MONTH,DAY,YEAR
REAL          R4
REAL          R8
CHARACTER(1)  C1
COMMON /NOALIGN/ R8,C1,R4     ! R4 will not be aligned on a
                              ! full-word boundary
```

関連情報

- 787 ページの『第 15 章 Pthreads ライブラリー・モジュール』
- 738 ページの『THREADLOCAL』
- 187 ページの『ブロック・データのプログラム単位』
- 86 ページの『明示的形状配列』
- グローバル・エンティティーの詳細については、 160 ページの『名前の有効範囲』
- 78 ページの『変数のストレージ・クラス』

COMPLEX

目的

COMPLEX タイプ宣言ステートメントは複素数タイプのオブジェクトと関数についての長さと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>—COMPLEX——————————————————————:: ——————————————————————entity_decl_list————————————><
           └kind_selector┘  └,—attr_spec_list—::┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

kind_selector

```
>>——┬—(——————————————int_initialization_expr—)——┬——————————————————————><
    │    └KIND— = ┘                              │
    │                            (1)             │
    └ * —int_literal_constant————————————————————┘
```

注:

1　IBM 拡張

これは複素数エンティティーの長さを指定します。

IBM 拡張

- *int_initialization_expr* を指定すると、有効な値は 4、8 および 16 になります。これらの値は複素数エンティティーの各部分の精度および範囲を表します。
- 形式 **int_literal_constant* を指定すると、有効値は 8、16 および 32 になります。これらの値は複素数エンティティー全体の長さを表し、代替形式に認められている値に対応します。 *int_literal_constant* には、kind 型付きパラメーターは指定できません。

IBM 拡張 の終り

attr_spec
特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
IN、**OUT**、または **INOUT** のいずれかです。

:: ダブル・コロン・セパレーターです。複数の属性を指定する場合、あるいは = *initialization_expr* F95 または **NULL()** F95 を使用する場合に必要になります。

array_spec
次元境界のリストです。

entity_decl

```
>>─a─┬──────────────────────────────────────────┬─────────────►
     ├─┬──────────────┬─┬───────────────────┬─┤
     │ │        (1)   │ └─(─array_spec─)─┘ │
     │ └─ * ─len──────┘                     │
     │                            (2)        │
     └─(─array_spec─)─ * ─len───────────────┘

►─┬────────────────────────────────┬──────────────────────►◄
  │                       (3)      │
  └─┬─/─initial_value_list─/─────┤
    ├─ = ─initialization_expr───┤
    │              (4)            │
    └─ => ─NULL()────────────────┘
```

注:

1 IBM 拡張

2 IBM 拡張

3 IBM 拡張

4 Fortran 95

a オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

len *kind_selector* に指定されている長さをオーバーライドします。kind 型付きパラメーターを指定することはできません。エンティティーの長さは、許容できる長さ指定の 1 つを表す整数のリテラル定数でなければなりません。

IBM 拡張 の終り

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するためにタイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。以下の場合にオブジェクトを初期化できます。

- オブジェクトが、ブロック・データ・プログラム単位内の名前付き共通ブロックにある。

IBM 拡張

- オブジェクトがモジュール内の名前付き共通ブロックにある。

IBM 拡張 の終り

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。*initialization_expr* F95 または **NULL()** F95 が指定されていて、

- エンティティーが変数の場合、変数が最初に定義されます。

Fortran 95

- エンティティーが派生型コンポーネントの場合、派生型にはデフォルトの初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。

変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、*a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている `T` または `F` がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

例

```
COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements
```

関連情報

- 33 ページの『複素数』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

CONTAINS

目的

CONTAINS ステートメントは、メインプログラム、外部サブプログラム、またはモジュール・サブプログラムの本体と、それらが含む内部サブプログラムを分離します。同様に、モジュールの仕様部分とモジュール・サブプログラムを分離します。

構文

```
►►—CONTAINS——————————————————————————————————————————————————————►◄
```

規則

CONTAINS ステートメントがある場合、その後に少なくとも 1 つのサブプログラムが続かなければなりません。

CONTAINS ステートメントはブロック・データ・プログラム単位にも内部サブプログラムにも置くことはできません。

CONTAINS ステートメントのラベルは、 **CONTAINS** ステートメントのあるメインプログラム、サブプログラム、またはモジュールの一部と考えられます。

例

```
MODULE A
   ⋮
  CONTAINS               ! Module subprogram must follow
  SUBROUTINE B(X)
```

```
    ⋮
     CONTAINS             ! Internal subprogram must follow
     FUNCTION C(Y)

    ⋮
     END FUNCTION
   END SUBROUTINE
 END MODULE
```

関連情報

169 ページの『プログラム単位、プロシージャー、およびサブプログラム』

CONTINUE

目的

CONTINUE ステートメントは何も処理を行わず、何の効果もない実行可能制御ステートメントです。このステートメントは、ループの終端ステートメントとしてよく使用されます。

構文

```
▶▶──CONTINUE──────────────────────────────────────────────▶◀
```

例

```
     DO 100 I = 1,N
        X = X + N
100  CONTINUE
```

関連情報

147 ページの『第 6 章 制御構造』

CYCLE

目的

CYCLE ステートメントは、 **DO** 構造体または **DO WHILE** 構造体の現行の実行サイクルを終了させます。

構文

```
▶▶──CYCLE──┬──────────────────┬──────────────────────────────────────▶◀
           └─DO_construct_name─┘
```

DO_construct_name
: **DO** または **DO WHILE** 構造体の名前です。

規則

CYCLE ステートメントは、**DO** 構造体または **DO WHILE** 構造体の中に置かれ、*DO_construct_name* によって指定された特定の **DO** または **DO WHILE** 構造体に所属します。*DO_construct_name* により特定の DO または DO WHILE 構造体が指定されていない場合は、このステートメントを直接包含する **DO** または **DO WHILE** 構造体に所属します。このステートメントは、属している構造体の現行のサイクルだけを終了させます。

CYCLE ステートメントを実行すると、**DO** または **DO WHILE** 構造体の現行の実行サイクルが終了します。 **CYCLE** ステートメント以降は、終了するラベル付きアクション・ステートメントを含め、すべての実行可能ステートメントは実行されません。 **DO** 構造体の場合、プログラムの実行は増分値の処理へと続きます。 **DO WHILE** 構造体の場合、プログラムの実行はループ制御処理へと続きます。

CYCLE ステートメントにはラベルを付けることができます。しかし、**DO** 構造体を終了させるラベル付きアクション・ステートメントとして使用することはできません。

例

```
LOOP1: DO I = 1, 20
   N = N + 1
   IF (N > NMAX) CYCLE LOOP1          ! cycle to LOOP1

   LOOP2: DO WHILE (K==1)
      IF (K > KMAX) CYCLE             ! cycle to LOOP2
      K = K + 1
   END DO LOOP2

   LOOP3:  DO J = 1, 10
      N = N + 1
      IF (N > NMAX) CYCLE LOOP1       ! cycle to LOOP1
      CYCLE LOOP3                     ! cycle to LOOP3
   END DO LOOP3

END DO LOOP1
END
```

関連情報

- 335 ページの『DO』
- 337 ページの『DO WHILE』

DATA

目的

DATA ステートメントは変数に初期値を与えます。

構文

```
        ┌─────────────────────────────────────────────┐
        │ └,┘                                         │
►►─DATA─▼─data_object_list─/─initial_value_list─/─────┴──────────────►◄
```

data_object

変数または *implied-DO list* です。添え字式またはサブストリング式は、初期化式でなければなりません。

implied-DO list

```
►►─(─do_object_list─,─do_variable─ = ─integer_expr1─,─integer_expr2──────────►

►─┬──────────────────┬─)──────────────────────────────────────────────────►◄
  └─,─integer_expr3──┘
```

do_object
: 配列エレメント、スカラー構造体コンポーネント、サブストリングまたは暗黙 **DO** リストです。

do_variable
: 暗黙 **DO** 変数と呼ばれる名前付きスカラー整変数です。この変数は、ステートメント・エンティティーです。

integer_expr1、*integer_expr2*、および *integer_expr3*
: スカラー整数式です。これらの式には、定数および別のいくつかの暗黙 **DO** リスト (当該の暗黙 **DO** リストをその範囲内に持つもの) の暗黙 **DO** 変数のみを含めることができます。各処理は組み込み型でなければなりません。

initial_value

```
>>-+-------+--data_value-----------------------------------------><
   '-r-- *-'
```

r

負でないスカラー整数定数です。 *r* が名前付き定数の場合、これは有効範囲単位内で事前に宣言されているか、あるいは使用関連付けまたはホスト関連付けによってアクセス可能にされている必要があります。

Fortran 95

また、*r* は、定数の負ではないスカラー整数サブオブジェクトです。上記と同様に、これが名前付き定数のサブオブジェクトである場合も、有効範囲単位内で事前に宣言されているか、あるいは使用関連付けまたはホスト関連付けによってアクセス可能にされている必要があります。

Fortran 95 の終り

r が定数のサブオブジェクトの場合、その中の添え字はどれも初期化式です。 *r* を省略すると、デフォルト値は 1 になります。 *r*data_value* という形式はデータ値を *r* 回連続して指定するのと同じです。

data_value

スカラー定数、符号付き整数リテラル定数、符号付き実リテラル定数、構造体コンストラクター、 F95 定数のスカラー・サブオブジェクト、または **NULL()** です。 F95

規則

data_object としてポインターでない配列オブジェクトを指定することは、配列オブジェクト内のすべてのエレメントのリストを格納された順に指定することと同じです。

Fortran 95

ポインター属性を持つ配列は、対応する初期値 **NULL()** を 1 つのみ持ちます。

Fortran 95 の終り

各 *data_object_list* は対応する *initial_value_list* と同じ数の項目を指定しなければなりません。 これらの 2 つのリストの項目は互いに 1 対 1 の対応をとります。この対応によって、それぞれの *data_object* の初期値が決定します。

Fortran 95

ポインターの初期化の場合、*data_value* が **NULL()** であれば、対応する *data_object* はポインター属性を持つことになります。 *data_object* がポインター属性を持つと、対応する *data_value* は **NULL()** になるはずです。

Fortran 95 の終り

initial_value による各 *data_object* の定義は、 66 ページの『タイプなし定数の使用方法』に記載している項目以外は組み込み割り当ての規則に従います。

initial_value が構造体コンストラクターの場合、各コンポーネントは初期化式でなければなりません。 *data_object* が変数の場合、サブストリング式または添え字式は初期化式でなければなりません。

data_value が名前付き定数または構造体コンストラクターの場合、その名前付き定数または派生型は、有効範囲単位内で事前に宣言されているか、あるいは使用関連付けまたはホスト関連付けによってアクセス可能にされている必要があります。

長さがゼロの文字変数はリストに変数を 1 つ与えますが、サイズがゼロの配列、反復カウントがゼロの暗黙 **DO** リスト、および反復係数がゼロの値は拡張 *initial_value_list* に何の値も与えません。

DATA ステートメント内で暗黙 **DO** リストを使用して配列エレメント、スカラー構造体構成子およびサブストリングを初期化できます。暗黙 **DO** 変数の制御の下で、暗黙 **DO** リストはスカラー構造体コンポーネント、配列エレメント、およびサブストリングの順序列に拡張されます。配列エレメントおよびスカラー構造体コンポーネントは、定数である親を持つことはできません。スカラー構造体コンポーネントはそれぞれ、添え字リストを指定するコンポーネントの参照を、少なくとも 1 つは含んでいなければなりません。

暗黙 **DO** リストの範囲は *do_object_list* です。 **DO** ステートメントと同様に、反復カウントおよび暗黙 **DO** 変数の値は、*integer_expr1*、 *integer_expr2*、および *integer_expr3* によって確立されます。暗黙 **DO** が実行されると、暗黙 **DO** が 1 回繰り返されるごとに、*do_object_list* 内の項目が指定され、その暗黙 **DO** 変数のその時点の値に応じた適切な値が割り当てられます。暗黙 **DO** 変数の反復カウントがゼロの場合、拡張順序列には何の変数も追加されません。

do_object 内の添え字式には、定数か、またはその範囲内に添え字式を持つ暗黙 **DO** リストの暗黙 **DO** 変数だけを入れることができます。各処理は組み込み型でなければなりません。

IBM 拡張

論理定数を持つ論理タイプのリスト項目を初期化する場合、省略形を使用することができます (.TRUE. の場合 T、 .FALSE. の場合 F)。 T または F が **PARAMETER** 属性によって事前に定義された定数名の場合、 XL Fortran はそのストリングを名前付き定数として認識し、その値を **DATA** ステートメント内の対応するリスト項目に割り当てます。

IBM 拡張 の終り

ブロック・データ・プログラム単位では、**DATA** ステートメントまたはタイプ宣言ステートメントを使用して初期値を名前付き共通ブロック内の変数に割り当てることができます。

内部またはモジュール・サブプログラムでは、*data_object* がホスト内のエンティティーと同じ名前を持ち、かつ内部サブプログラム内の他の仕様ステートメントで *data_object* が宣言されていない場合、**DATA** ステートメントの前では *data_object* を参照することも定義することもできません。

DATA ステートメントは以下のものに初期値を割り当てることはできません。
- 自動オブジェクト。
- 仮引き数。
- IBM ポインティング先。 IBM
- 無名共通ブロック内の変数。
- 関数の結果変数。
- IBM ストレージ・クラスが自動であるデータ・オブジェクト。 IBM
- **ALLOCATABLE** 属性を持つ変数。

実行可能プログラム内では何度も変数を初期化することはできません。複数の変数を共用する場合、データ・オブジェクトのうち 1 つだけを初期化できます。

例

例 1:

```
      INTEGER Z(100),EVEN_ODD(0:9)
      LOGICAL FIRST_TIME
      CHARACTER*10 CHARARR(1)
      DATA    FIRST_TIME / .TRUE. /
      DATA    Z  / 100* 0 /
! Implied-DO list
      DATA  (EVEN_ODD(J),J=0,8,2) / 5 * 0 /  &
    &      ,(EVEN_ODD(J),J=1,9,2) / 5 * 1 /
! Nested example
      DIMENSION TDARR(3,4)  ! Initializes a two-dimensional array
      DATA ((TDARR(I,J),J=1,4),I=1,3) /12 * 0/
! Character substring example
```

```
      DATA (CHARARR(J)(1:3),J=1,1) /'aaa'/
      DATA (CHARARR(J)(4:7),J=1,1) /'bbbb'/
      DATA (CHARARR(J)(8:10),J=1,1) /'ccc'/
! CHARARR(1) contains 'aaabbbbccc'
```

例 2:

```
TYPE DT
      INTEGER :: COUNT(2)
END TYPE DT

TYPE(DT), PARAMETER, DIMENSION(3) :: SPARM = DT ( (/3,5/) )

INTEGER :: A(5)

DATA A /SPARM(2)%COUNT(2) * 10/
```

関連情報

- 27 ページの『第 3 章 データ型およびデータ・オブジェクト』
- 154 ページの『DO ステートメントの実行』
- 163 ページの『ステートメント・エンティティーおよび構造体エンティティー』

DEALLOCATE

目的

DEALLOCATE ステートメントは、割り振り可能オブジェクトとポインター・ターゲットを動的に割り振り解除します。ターゲットと関連する他のポインターが未定義である場合、指定したポインターとの関連は解除されます。

構文

```
►►──DEALLOCATE──(──allocate_object_list──┬──────────────────────────┬──)──────────►◄
                                         └─,──STAT── = ──stat_variable─┘
```

object ポインターまたは割り振り可能オブジェクトです。

stat_variable
スカラー整変数です。

規則

DEALLOCATE ステートメントで指定する割り振り可能オブジェクトは、現在割り振り済みでなければなりません。関連するポインターを介して、**TARGET** 属性を持つ割り振り可能オブジェクトの割り振りを解除することはできません。そのようなオブジェク

トの割り振りを解除すると、関連するポインターの関連付け状況は未定義になります。未定義の割り振り状況を持つ割り振り可能オブジェクトに対して、その時点で、参照、定義、割り振り、または割り振り解除を行うことはできません。 **DEALLOCATE** ステートメントが正常に実行されると、割り振り可能オブジェクトの割り振り状況は、割り振られていないという状況になります。

派生型の変数の割り振りが解除されると、割り振り済みサブオブジェクトの割り振りも解除されます。

組み込み割り当てステートメントが実行されると、割り当てが行われる前に、変数の割り振り済みサブオブジェクトの割り振りが解除されます。

DEALLOCATE ステートメント内のポインターは、**ALLOCATE** ステートメントで作成されたターゲット全体と関連を持たなければなりません。ポインター・ターゲットの割り振りを解除すると、ターゲット全体あるいは一部と関連を持つ他のポインター関連付け状況は未定義になります。

ヒント

割り振り済みストレージに関連したポインターが他にない場合は、**NULLIFY** ではなく **DEALLOCATE** ステートメントを使用してください。

ポインター関数が割り振られたストレージの割り振りを解除してください。

STAT= 指定子を指定せず、このステートメントの実行中にエラーが発生した場合、プログラムは終了します。 **STAT=** 指定子を指定した場合、*stat_variable* には以下の値の 1 つが割り当てられます。

IBM 拡張

Stat 値	エラー状態
0	エラーなし
1	割り振り解除を試みているシステム・ルーチンにエラー
2	割り振り解除に無効なデータ・オブジェクトが指定された
3	1 と 2 の両方のエラーが発生した

IBM 拡張 の終り

allocate_object は、同じ DEALLOCATE ステートメント内の別の *allocate_object* の値、境界、割り振り状況、または関連付け状況に依存せず、また、同じ DEALLOCATE ステートメント内の *stat_variable* の値にも依存しません。

stat_variable は、同じ DEALLOCATE ステートメント内で割り振りを解除することはできません。また、同じ DEALLOCATE ステートメント内の *allocate_object* の値、境界、割り振り状況、または関連付け状況に依存しません。

例

```
INTEGER, ALLOCATABLE :: A(:,:)
INTEGER X,Y
   ⋮
ALLOCATE (A(X,Y))
   ⋮
DEALLOCATE (A,STAT=I)
END
```

関連情報

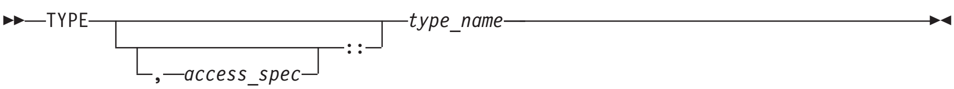
- 290 ページの『ALLOCATE』
- 289 ページの『ALLOCATABLE』
- 77 ページの『割り振り状況』
- 167 ページの『ポインター関連付け』
- 90 ページの『据え置き形状配列』
- 203 ページの『仮引き数として割り振り可能なオブジェクト』
- 51 ページの『割り振り可能コンポーネント』

派生型 (TYPE)

目的

派生型 (TYPE) ステートメントは派生型定義の最初のステートメントです。

構文



access_spec
: **PRIVATE** または **PUBLIC** です。

type_name
: 派生型の名前です。

規則

access_spec は、派生型定義がモジュールの仕様部分にあるときにだけ指定できます。

type_name は、**BYTE** 以外のどの組み込みタイプ、また他のアクセス可能派生型と同じ名前にすることはできません。

派生型 (TYPE) ステートメントでラベルを指定すると、そのラベルは派生型定義の有効範囲単位に属します。

対応する **END TYPE** ステートメントで名前を指定する場合、その名前は *type_name* と同じでなければなりません。

例

```
MODULE ABC
  TYPE, PRIVATE :: SYSTEM      ! Derived type SYSTEM can only be accessed
    SEQUENCE                   !   within module ABC
    REAL :: PRIMARY
    REAL :: SECONDARY
    CHARACTER(20), DIMENSION(5) :: STAFF
  END TYPE
END MODULE
```

関連情報

- 41 ページの『派生型』
- 356 ページの『END TYPE』
- 467 ページの『SEQUENCE』
- 438 ページの『PRIVATE』

DIMENSION

目的

DIMENSION 属性は配列の名前と次元を指定します。

構文

```
>>─DIMENSION─┬────┬─array_declarator_list─────────────────────────><
             └─::─┘
```

規則

Fortran 95 では、配列の次元は 7 まで指定することができます。

IBM 拡張

XL Fortran では、配列の次元は 20 まで指定することができます。

IBM 拡張 の終り

1 つの有効範囲単位内では、1 つの配列名について 1 回しか次元指定を実行できません。

DIMENSION 属性と互換性のある属性

- ALLOCATABLE
- AUTOMATIC
- INTENT
- OPTIONAL
- PARAMETER
- POINTER
- PRIVATE
- PROTECTED
- PUBLIC
- SAVE
- STATIC
- TARGET
- VOLATILE

例

```
CALL SUB(5,6)
CONTAINS
SUBROUTINE SUB(I,M)
  DIMENSION LIST1(I,M)                         ! automatic array
  INTEGER, ALLOCATABLE, DIMENSION(:,:) :: A    ! deferred-shape array
    ⋮
END SUBROUTINE
END
```

関連情報

- 83 ページの『第 4 章 配列の概念』
- 493 ページの『VIRTUAL』

DO

目的

DO ステートメントは、それ自身と、指定された終端ステートメントまでの間にあるステートメント (その終端ステートメントも含む) の実行を制御します。これらのステートメントすべてが **DO** 構造体を構成します。

構文

```
►►──┬───────────────────────┬──DO──┬────────────┬──────────────────────►
    └─DO_construct_name──:──┘      └─stmt_label─┘

►──┬─────────────────────────────────────────────────────┬────────────►◄
   └─┬───┬─var_name = a_expr1, a_expr2─┬──────────────┬──┘
     └─,─┘                             └─ , a_expr3───┘
```

DO_construct_name
: **DO** 構造体を識別する名前です。

stmt_label
: 同一の有効範囲単位内の **DO** ステートメントの後に指定されている実行可能ステートメントのラベルです。このステートメントは、**DO** 構造体の終わりを示します。

var_name
: 整数タイプまたは実数タイプのスカラー変数名で、**DO** 変数と呼ばれます。

a_expr1、*a_expr2*、**および** *a_expr3*
: これらは、整数タイプまたは実数タイプのスカラー式です。

規則

DO ステートメントで *DO_construct_name* を指定する場合、同じ *DO_construct_name* が指定されている **END DO** でその構造体を終了させます。逆に、**DO** ステートメント上に *DO_construct_name* を指定せずに、 **END DO** で **DO** 構造体を終了させる場合は、 **END DO** ステートメントに *DO_construct_name* を指定してはなりません。

DO ステートメントでステートメント・ラベルを指定した場合は、同じステートメント・ラベルの付いたステートメントで **DO** 構造体を終了させなければなりません。ラベル付き **DO** ステートメントを同じラベルが付いている **END DO** ステートメントで終了させることはできますが、ラベルなし **END DO** ステートメントで終了させることはできません。 **DO** ステートメントにラベルを指定しない場合は、**END DO** ステートメントで **DO** 構造体を終了させなければなりません。

制御文節 (*var_name* で始まる文節) を指定しないと、そのステートメントは無限 **DO** になります。ループは (たとえば **EXIT** ステートメントなどによって) 中断されるまで、いつまでも実行を繰り返します。

例

```
INTEGER :: SUM=0
OUTER: DO
  INNER: DO M=1,10
    READ (5,*) J
    IF (J.LE.I) THEN
```

```
      PRINT *, 'VALUE MUST BE GREATER THAN ', I
      CYCLE INNER
    END IF
    SUM=SUM+J
    IF (SUM.GT.500) EXIT OUTER
    IF (SUM.GT.100) EXIT INNER
  END DO INNER
  SUM=SUM+I
  I=I+10
END DO OUTER
PRINT *, 'SUM =',SUM
END
```

関連情報

- 152 ページの『DO 構造体』
- **END DO** ステートメントの詳細については、 351 ページの『END (構造体)』
- 365 ページの『EXIT』
- 325 ページの『CYCLE』
- 519 ページの『INDEPENDENT』
- 511 ページの『ASSERT』
- 513 ページの『CNCALL』
- 525 ページの『PERMUTATION』
- 718 ページの『PARALLEL DO / END PARALLEL DO』

DO WHILE

目的

DO WHILE ステートメントは、**DO WHILE** 構造体の最初のステートメントであり、指定された終端ステートメントまでの間にある (終端ステートメントも含む) のステートメントのブロックを、このステートメントに指定されている論理式が真であり続ける限り、繰り返し実行します。

構文

```
>>─┬─────────────────────────┬─DO─┬────────────┬─┬───┬─WHILE─(─logical_expr─)──><
   └─DO_construct_name─:─┘      └─stmt_label─┘ └─,─┘
```

DO_construct_name
: **DO WHILE** 構造体を識別する名前です。

stmt_label
: 同一の有効範囲単位内の **DO WHILE** ステートメントの後に指定されている実行可能ステートメントのラベルです。これは **DO WHILE** 構造体の終わりを示します。

logical_expr
スカラー論理式です。

規則

DO WHILE ステートメントで *DO_construct_name* を指定する場合、同じ *DO_construct_name* が指定されている **END DO** でその構造体を終了させます。逆に、**DO WHILE** ステートメント上に *DO_construct_name* を指定せずに、**END DO** で **DO WHILE** 構造体を終了させる場合は、**END DO** ステートメントに *DO_construct_name* を指定してはなりません。

DO WHILE ステートメントにラベルを指定する場合、同じラベルの付いたステートメントで **DO WHILE** 構造体を終了させなければなりません。ラベル付き **DO WHILE** ステートメントを同じラベルが付いている **END DO** ステートメントで終了させることはできますが、ラベルなし **END DO** ステートメントで終了させることはできません。**DO WHILE** ステートメントにラベルを指定しない場合は、**END DO** ステートメントで **DO WHILE** 構造体を終了させなければなりません。

例

```
      MYDO: DO 10 WHILE (I .LE. 5)  ! MYDO is the construct name
         SUM = SUM + INC
         I = I + 1
10    END DO MYDO
      END

      SUBROUTINE EXAMPLE2
        REAL X(10)
        LOGICAL FLAG1
        DATA    FLAG1 /.TRUE./
        DO 20 WHILE (I .LE. 10)
           X(I) = A
           I = I + 1
20      IF (.NOT. FLAG1) STOP
      END SUBROUTINE EXAMPLE2
```

関連情報

- 157 ページの『DO WHILE 構造体』
- **END DO** ステートメントの詳細については、351 ページの『END (構造体)』
- 365 ページの『EXIT』
- 325 ページの『CYCLE』

DOUBLE COMPLEX

IBM 拡張

目的

DOUBLE COMPLEX タイプ宣言ステートメントは倍精度複素数タイプのオブジェクトと関数の長さと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>──DOUBLE COMPLEX──┬─::──────────────────────┬─entity_decl_list──────────────><
                    └─,──attr_spec_list──::───┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

attr_spec
: 特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
: **IN**、**OUT**、または **INOUT** のいずれかです。

::
: ダブル・コロン・セパレーターです。複数の属性を指定するとき、または = *initialization_expr* あるいは => **NULL()** を使用するときに必要になります。

array_spec
: 次元境界のリストです。

entity_decl

```
►►─a─┬──────────────────┬──┬─────────────────────────────┬────────────────────►◄
     └─(─array_spec─)───┘  ├─/─initial_value_list─/──────┤
                           ├─=─initialization_expr───────┤
                           └─ => ─NULL()─────────────────┘
```

a	オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。
initial_value	直前の名前によって指定されるエンティティーに初期値を与えます。
initialization_expr	初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。
=> **NULL()**	ポインター・オブジェクトに初期値を与えます。

規則

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することは

できません。ブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、またはモジュール内の名前付き共通ブロックにある場合、オブジェクトは初期化することができます。

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、 *initialization_expr* または **NULL()** を指定した場合、変数は最初に定義されます。宣言するエンティティーが派生型のコンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルトの初期化があります。 *a* は、組み込み割り当ての規則に従って、 *initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、同じ有効範囲単位内のタイプ宣言ステートメントか前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。 *a* が変数で、 *initialization_expr* か => **NULL()** がある場合、名前付き共通ブロックの中のオブジェクト以外は、*a* が保管済みオブジェクトであることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

事前に定数の名前として定義されている T または F がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

例

```
SUBROUTINE SUB
  DOUBLE COMPLEX, STATIC, DIMENSION(1) :: B
END SUBROUTINE
```

関連情報

- 33 ページの『複素数』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

IBM 拡張 の終り

DOUBLE PRECISION

目的

DOUBLE PRECISION タイプ宣言ステートメントは倍精度タイプのオブジェクトと関数の長さと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
►►──DOUBLE PRECISION──┬─────────────────────┬──entity_decl_list──────────►◄
                      └─,──attr_spec_list──::┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

attr_spec
特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
IN、**OUT**、または **INOUT** のいずれかです。

:: ダブル・コロン・セパレーターです。複数の属性を指定するとき、または = *initialization_expr* あるいは F95 => **NULL()** F95 を使用するときに必要になります。

array_spec
次元境界のリストです。

entity_decl

```
▶▶─a─┬──────────────────┬─┬─────────────────────────────────┬─────────────▶◀
     └─(─array_spec─)─┘ │                (1)              │
                        ├─/─initial_value_list─────────/─┤
                        ├─ = ─initialization_expr─────────┤
                        │              (2)                │
                        └─ => ─NULL()─────────────────────┘
```

注:

1 IBM 拡張

2 Fortran 95

a オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。オブジェクトがブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、 IBM またはモジュール内の名前付き共通ブロックにある場合、

IBM そのオブジェクトを初期化することができます。

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

Fortran 95

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、*initialization_expr* または **NULL()** を指定した場合、変数は最初に定義されます。宣言するエンティティーが派生型のコンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルトの初期化があります。*a* は、組み込み割り当ての規則に従って、 *initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、同じ有効範囲単位内のタイプ宣言ステートメントか前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。 *a* が変数で、 *initialization_expr* か => **NULL()** がある場合、名前付き共通ブロックの中のオブジェクト以外は、*a* が保管済みオブジェクトであることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与

える場合もあります。

Fortran 95 の終り

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている T または F がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

例

```
DOUBLE PRECISION, POINTER :: PTR
DOUBLE PRECISION, TARGET  :: TAR
```

関連情報

- 31 ページの『実数』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

ELSE

目的

ELSE ステートメントは **IF** 構造体内のオプションの **ELSE** ブロックの最初のステートメントです。

構文

```
►►──ELSE──┬──────────────────┬──────────────────────────────────────────────►◄
          └─IF_construct_name─┘
```

IF_construct_name
　　IF 構造体を識別する名前です。

構文

IF 構造体内で前に存在するすべての論理式が偽の場合、制御は **ELSE** に分岐します。**ELSE** ブロックのステートメント・ブロックが実行され、**IF** 構造体が完了します。

IF_construct_name を指定した場合、その名前はブロック **IF** ステートメントに指定した名前と同じでなければなりません。

例

```
IF (A.GT.0) THEN
  B = B-A
ELSE            ! the next statement is executed if a<=0
  B = B+A
END IF
```

関連情報

- 147 ページの『IF 構造体』
- **END IF** ステートメントの詳細については、 351 ページの『END (構造体)』
- 『ELSE IF』

ELSE IF

目的

ELSE IF ステートメントは **IF** 構造体内のオプションの **ELSE IF** ブロックの最初のステートメントです。

構文

```
►►──ELSE IF──(──scalar_logical_expr──)──THEN──┬──────────────────┬──────────►◄
                                              └─IF_construct_name─┘
```

IF_construct_name
　　IF 構造体を識別する名前です。

規則

IF 構造体内で前にあるどの論理式も真ではない場合、 *scalar_logical_expr* が計算されます。 *scalar_logical_expr* が真の場合、それに続くステートメント・ブロックが実行され、**IF** 構造体が完了します。

IF_construct_name を指定した場合、その名前はブロック **IF** ステートメントに指定した名前と同じでなければなりません。

例

```
IF (I.EQ.1) THEN
    J=J-1
ELSE IF (I.EQ.2) THEN
    J=J-2
ELSE IF (I.EQ.3) THEN
    J=J-3
ELSE
    J=J-4
END IF
```

関連情報

- 147 ページの『IF 構造体』
- **END IF** ステートメントの詳細については、 351 ページの『END (構造体)』
- 346 ページの『ELSE』

ELSEWHERE

目的

ELSEWHERE ステートメントは **WHERE** 構造体内のオプションの **ELSEWHERE** ブロック、またはマスクされた **ELSEWHERE** ブロックの最初のステートメントです。

構文

```
>>—ELSEWHERE—┬──────────────────────┬─┬─────────────────────────────┬──────────><
             │                (1)   │ │                        (2)  │
             └─(—mask_expr————)─────┘ └─where_construct_name────────┘
```

注:

1 Fortran 95

2 Fortran 95

Fortran 95

mask_expr　　論理配列式です。

Fortran 95 の終り

Fortran 95

where_construct_name
WHERE 構造体を識別する名前です。

Fortran 95 の終り

規則

Fortran 95

マスクされた **ELSEWHERE** ステートメントには *mask_expr* が含まれます。マスク式の解釈については、 133 ページの『マスクされた配列割り当ての解釈』 を参照してください。 **WHERE** 構造体内のそれぞれの *mask_expr* は同じ形状でなければなりません。

where_construct_name を指定する場合、その名前は **WHERE** 構造体ステートメントに指定した名前と同じでなければなりません。

Fortran 95 の終り

ELSEWHERE およびマスクされた **ELSEWHERE** ステートメントは、分岐ターゲット・ステートメントにすることはできません。

例

以下の例では、単純なマスクされた **ELSEWHERE** ステートメントを使って配列内のデータを変更するプログラムを示しています。

```
INTEGER ARR1(3, 3), ARR2(3, 3), FLAG(3, 3)

ARR1 = RESHAPE((/(I, I=1, 9)/), (/3, 3 /))
ARR2 = RESHAPE((/(I, I=9, 1, -1 /), (/3, 3 /))
FLAG = -99

! Data in arrays ARR1, ARR2, and FLAG at this point:
!
! ARR1 = |   1   4   7 |  ARR2 = |   9   6   3 |  FLAG = | -99 -99 -99 |
!        |   2   5   8 |         |   8   5   2 |         | -99 -99 -99 |
!        |   3   6   9 |         |   7   4   1 |         | -99 -99 -99 |

WHERE (ARR1 > ARR2)
  FLAG = 1
ELSEWHERE (ARR1 == ARR2)
  FLAG = 0
```

```
ELSEWHERE
  FLAG = -1
END WHERE

! Data in arrays ARR1, ARR2, and FLAG at this point:
!
! ARR1 = |   1   4   7 |  ARR2 = |   9   6   3 |  FLAG = | -1  -1   1 |
!        |   2   5   8 |         |   8   5   2 |         | -1   0   1 |
!        |   3   6   9 |         |   7   4   1 |         | -1   1   1 |
```

関連情報

- 131 ページの『WHERE 構造体』
- 497 ページの『WHERE』
- **END WHERE** ステートメントの詳細については、 351 ページの『END (構造体)』

END

目的

END ステートメントは、プログラム単位またはプロシージャーの終了を示します。

構文

```
>>--END--+-------------------------------------+------------------------><
         +-BLOCK DATA--+-------------------+---+
         |             '-BLOCK_DATA_name---'   |
         +-FUNCTION--+-----------------+-------+
         |           '-FUNCTION_name---'       |
         +-MODULE--+---------------+-----------+
         |         '-MODULE_name---'           |
         +-PROGRAM--+----------------+---------+
         |          '-PROGRAM_name---'         |
         '-SUBROUTINE--+-------------------+---'
                       '-SUBROUTINE_name---'
```

規則

END ステートメントは、プログラム単位内で唯一の必須ステートメントです。

内部サブプログラムまたはモジュール・サブプログラムの場合、 **FUNCTION** または **SUBROUTINE** キーワードを **END** ステートメントで指定する必要があります。ブロック・データ・プログラム単位、外部サブプログラム、メインプログラム、モジュールおよびインターフェース本体の場合、対応するキーワードはオプションとなります。

オプションの **PROGRAM** ステートメントを使用し、プログラム名が **PROGRAM** ステートメントで指定したプログラム名と一致する場合にのみ、そのプログラム名を **END PROGRAM** ステートメントに含めることができます。

ブロック・データ名が **BLOCK DATA** ステートメント内で与えられ、 **BLOCK DATA** ステートメントに指定したプログラム名と一致する場合にのみ、そのブロック・データ名を **END BLOCK DATA** ステートメントに含めることができます。

END MODULE、**END FUNCTION**、または **END SUBROUTINE** ステートメントに名前を指定する場合、その名前はそれぞれ **MODULE**、**FUNCTION**、または **SUBROUTINE** ステートメントに指定されているものと同じでなければなりません。

END、**END FUNCTION**、**END PROGRAM**、および **END SUBROUTINE** ステートメントは、分岐可能な実行可能ステートメントです。固定形式および Fortran 90 自由形式の書式では、1 つの行で **END** ステートメントの後に他のステートメントを続けることはできません。固定形式の書式では、プログラム単位の **END** ステートメントを継続することはできません。また、開始行がプログラム単位の **END** ステートメントになるステートメントも継続できません。

メインプログラムの **END** ステートメントは、プログラムの実行を終了させます。関数またはサブルーチンの **END** には、 **RETURN** ステートメントと同じ機能があります。インライン注釈を、 **END** ステートメントと同じ行に指定することができます。 **END** ステートメントの後の注釈行は次のプログラム単位に属します。

例

```
PROGRAM TEST
  CALL SUB()
  CONTAINS
    SUBROUTINE SUB
   ⋮
    END SUBROUTINE    ! Reference to subroutine name SUB is optional
END PROGRAM TEST
```

関連情報

159 ページの『第 7 章 プログラム単位およびプロシージャー』

END (構造体)

目的

END DO、**END IF**、**END SELECT**、および **END WHERE** ステートメントは、それぞれ **DO** (または **DO WHILE**)、 **IF**、**CASE**、および **WHERE** 構造体を終了させます。

Fortran 95

END FORALL ステートメントは **FORALL** 構造体を終了させます。

Fortran 95 の終り

構文

```
>>-+-END DO--+-------------------+---------------+------------><
   |         '-DO_construct_name-'               |
   |                                   (1)       |
   +-END FORALL--+-----------------------+-------+
   |             '-FORALL_construct_name-'       |
   +-END IF--+-------------------+---------------+
   |         '-IF_construct_name-'               |
   +-END SELECT--+---------------------+---------+
   |             '-CASE_construct_name-'         |
   |                                 (2)         |
   '-END WHERE--+----------------------+---------'
                '-where_construct_name-'
```

注:

1 Fortran 95

2 Fortran 95

DO_construct_name
DO または **DO WHILE** 構造体を識別する名前です。

Fortran 95

FORALL_construct_name
FORALL 構造体を識別する名前です。

Fortran 95 の終り

IF_construct_name
IF 構造体を識別する名前です。

CASE_construct_name
CASE 構造体を識別する名前です。

Fortran 95

where_construct_name

WHERE 構造体を識別する名前です。

Fortran 95 の終り

規則

END DO ステートメントにラベルを付けると、ラベル付きまたはラベルなしの **DO** または **DO WHILE** 構造体の終端ステートメントとして使用することができます。 **END DO** ステートメントが終了させる構造体は最も内側の **DO** または **DO WHILE** 構造体だけです。 **DO** または **DO WHILE** ステートメントがステートメント・ラベルを指定しない場合、**DO** または **DO WHILE** 構造体の終端ステートメントは **END DO** ステートメントでなければなりません。

DO (または **DO WHILE**)、**IF**、または **CASE** 構造体の内部から、それぞれ **END DO**、**END IF**、または **END SELECT** ステートメントに分岐できます。 **END IF** ステートメントには **IF** 構造体の外部からも分岐できます。

Fortran 95

Fortran 95 では、 **END IF** ステートメントには **IF** 構造体の外部からは分岐できません。

Fortran 95 の終り

構造体の最初のステートメントに構造体名を指定した場合、構造体を終了させる **END** ステートメントは同じ構造体名を持っていなければなりません。構造体の最初のステートメントに構造体名を指定した場合、構造体を終了させる **END** ステートメントは同じ構造体名を持っていなければなりません。

END WHERE ステートメントは分岐ターゲット・ステートメントにはなれません。

例

```
INTEGER X(100,100)
DECR: DO WHILE (I.GT.0)
    ⋮
  IF (J.LT.K) THEN
    ⋮
  END IF                ! Cannot reference a construct name
  I=I-1
END DO DECR             ! Reference to construct name DECR mandatory

END
```

以下の例は、無効な *where_construct_name* の使用を示しています。

```
BW: WHERE (A /= 0)
  B = B + 1
END WHERE EW      ! The where_construct_name on the END WHERE statement
                  ! does not match the where_construct_name on the WHERE
                  ! statement.
```

関連情報

- 147 ページの『第 6 章 制御構造』
- 335 ページの『DO』
- 367 ページの『FORALL』
- 371 ページの『FORALL (構造体)』
- 387 ページの『IF (ブロック)』
- 465 ページの『SELECT CASE』
- 497 ページの『WHERE』
- 927 ページの『削除された機能』

END INTERFACE

目的

END INTERFACE ステートメントはプロシージャー・インターフェース・ブロックを終了させます。

構文

```
>>─END INTERFACE─┬─────────────────────┬──────────────────────────><
                 │                (1)  │
                 └─generic_spec────────┘
```

注:

1　　Fortran 95

Fortran 95

generic_spec

```
>>─┬─generic_name──────────────────────┬──────────────────────────><
   ├─OPERATOR─(─defined_operator─)─────┤
   └─ASSIGNMENT─(─ = ─)────────────────┘
```

Fortran 95 の終り

Fortran 95

defined_operator
: 定義された単項演算子、定義された 2 進演算子、または拡張組み込み演算子です。

Fortran 95 の終り

規則

INTERFACE ステートメントにはそれぞれ、対応する **END INTERFACE** ステートメントが必要です。

Fortran 95

END INTERFACE ステートメントに *generic_spec* を指定する場合、それは **INTERFACE** ステートメント内の対応する *generic_spec* と一致しなければなりません。

Fortran 95 の終り

END INTERFACE ステートメントに *generic_spec* を指定しない場合、*generic_spec* のあるなしに関係なく、すべての **INTERFACE** ステートメントと一致させることができます。

Fortran 95

END INTERFACE ステートメント内の *generic_spec* が *generic_name* である場合、対応する **INTERFACE** ステートメントの *generic_spec* は、同じ *generic_name* でなければなりません。

END INTERFACE ステートメント内の *generic_spec* が **OPERATOR**(*defined_operator*) である場合、対応する **INTERFACE** ステートメントの *generic_spec* は、同じ **OPERATOR**(*defined_operator*) でなければなりません。

END INTERFACE ステートメント内の *generic_spec* が **ASSIGNMENT**(=) である場合、対応する **INTERFACE** ステートメントの *generic_spec* は、同じ **ASSIGNMENT**(=) でなければなりません。

Fortran 95 の終り

例

```
INTERFACE OPERATOR (.DETERMINANT.)
  FUNCTION DETERMINANT (X)
    INTENT(IN) X
    REAL X(50,50), DETERMINANT
  END FUNCTION
END INTERFACE
```

Fortran 95

```
INTERFACE OPERATOR(.INVERSE.)
  FUNCTION INVERSE(Y)
    INTENT(IN) X
    REAL Y(50,50), INVERSE
  END FUNCTION
END INTERFACE OPERATOR(.INVERSE.)
```

Fortran 95 の終り

関連情報

- 407 ページの『INTERFACE』
- 170 ページの『インターフェースの概念』

END TYPE

目的

END TYPE ステートメントは、派生型の定義の完了を示します。

構文

```
>>—END TYPE—┬───────────┬——————————————————————————————————————————><
            └─type_name─┘
```

規則

type_name を指定した場合、その名前は対応する**派生型 (TYPE)** ステートメント内の *type_name* と一致しなければなりません。

END TYPE ステートメントにラベルを指定すると、そのラベルは派生型定義の有効範囲単位に属します。

例

```
TYPE A
  INTEGER :: B
  REAL :: C
END TYPE A
```

関連情報

- 41 ページの『派生型』
- 333 ページの『派生型 (TYPE)』

ENDFILE

目的

ENDFILE ステートメントは、順次アクセス用に接続された外部ファイルの次のレコードとしてファイルの最後のレコードを書き込みます。このレコードはファイルの最後のレコードになります。

ストリーム・アクセス用に接続されたファイルでは、**ENDFILE** ステートメントは、終端点を現在のファイル位置にします。現在の位置より前のファイル記憶単位は書き込み済みであると見なされ、読み取ることができます。ストリーム出力ステートメントを続けて使用することによって、さらにデータをファイルに書き込むことができます。

構文

```
►►──ENDFILE──┬─u───────────────────┬──────────────────────────────────►◄
             └─(──position_list──)─┘
```

u 外部装置識別子です。 *u* の値はアスタリスク、またはホレリス定数であってはなりません。

position_list
装置指定子 ([**UNIT=**]*u*) を必ず 1 つ含んでいなければならないリストです。このリストには、他の有効な各指定子を 1 つずつ入れることができます。

[UNIT=] *u*
装置指定子です。*u* は外部装置識別子で、その値はアスタリスクであってはなりません。外部装置識別子はスカラー整数式 (1 〜 2147483647 の値を持つ) で表される外部ファイルを示します。オプションの文字である **UNIT=** を省略する場合は、*u* は *position_list* の最初の項目でなければなりません。

IOSTAT= *ios*
I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのス

カラー変数またはデフォルトの整数です。 **ENDFILE** ステートメントの実行が完了すると、*ios* の値は次のように定義されます。
- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値

ERR= *stmt_label*
エラーが発生した場合に制御が移される、同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子は、エラー・メッセージを抑制します。

規則

IBM 拡張

装置が接続されていない場合、**fort.***n* という名前のデフォルトのファイルに対して順次アクセスを指定する暗黙の **OPEN** ステートメントが実行されます。ここで、*n* は先行ゼロを除去した *u* の値です。

順次アクセス用に接続されたファイルに対して **ENDFILE** ステートメントを実行した後は、データ転送入出力ステートメントまたは **ENDFILE** ステートメントを実行する前に、**BACKSPACE** または **REWIND** ステートメントを使用してファイルの位置を指定し直す必要があります。

2 つの **ENDFILE** ステートメントを同一のファイルに対して実行した場合、それらの間に **REWIND** または **BACKSPACE** ステートメントがなければ、2 番目の **ENDFILE** ステートメントは無視されます。

IBM 拡張 の終り

ERR= と **IOSTAT=** 指定子が設定されているときにエラーが検出されると、**ERR=** 指定子によって指定されたステートメントに対して転送が行われ、正の整数値が *ios* に割り当てられます。

IBM 拡張

IOSTAT= も **ERR=** も指定していない場合は、以下のとおりです。
- 重大なエラーが検出されるとプログラムは停止します。
- 回復可能エラーが検出され、 **ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、プログラムは次のステートメントの処理を継続します。オプションが **NO** に設定されていると、プログラムは停止します。

IBM 拡張 の終り

例

```
ENDFILE 12
ENDFILE (IOSTAT=IOSS,UNIT=11)
```

関連情報

- 229 ページの『条件および IOSTAT 値』
- 217 ページの『第 8 章 I/O の概念』
- 「ユーザーズ・ガイド」の『*実行時オプションの設定*』

ENTRY

目的

関数サブプログラムまたはサブルーチン・サブプログラムには、 **SUBROUTINE** または **FUNCTION** ステートメントを介して設定される 1 次エントリー・ポイントがあります。 **ENTRY** ステートメントは外部サブプログラムまたはモジュール・サブプログラムの代替エントリー・ポイントを設定します。

構文

```
>>--ENTRY--entry_name------------------------------------------------------>

>--+------------------------------------------------------------------+--><
   '-(--+---------------------+--)--+-------------------------------+-'
        '-dummy_argument_list-'     '-RESULT--(--result_name--)-'
```

entry_name

関数サブプログラムまたはサブルーチン・サブプログラムの中のエントリー・ポイントの名前です。

規則

ENTRY ステートメントを、メインプログラム、 ブロック・データ・プログラム単位、内部サブプログラム、**IF** 構造体、**DO** 構造体、 **CASE** 構造体、派生型定義、またはインターフェース・ブロックの中に指定することはできません。

IBM 拡張

ENTRY ステートメントは、 **CRITICAL**、**MASTER**、**PARALLEL**、**PARALLEL SECTIONS**、**SECTIONS**、または **SINGLE** 構造体内に指定できません。

IBM 拡張 の終り

ENTRY ステートメントは、外部サブプログラムまたはモジュール・サブプログラムの **FUNCTION** または **SUBROUTINE** ステートメント (および **USE** ステートメントの後) であれば、どこに指定してもかまいません。ただし、制御構造体内のステートメント・ブロックの内部、派生型定義の内部、インターフェース・ブロックの内部に指定することはできません。**ENTRY** ステートメントは非実行可能ステートメントであるため、サブプログラム実行中の制御順序には影響を与えません。

結果変数を指定している場合、その値は *result_name* です。指定していない場合は、*entry_name* になります。 **ENTRY** ステートメントの結果変数の特性が **FUNCTION** ステートメントの結果変数の特性と同じ場合、それらの結果変数は、たとえ名前が違っていても、同じ変数を識別します。それ以外の場合は、結果変数は同じストレージを共用し、すべてが組み込み (非文字) タイプの非ポインター・スカラーになります。*result_name* は、**FUNCTION** ステートメントまたは別の **ENTRY** ステートメントに対して指定された結果変数と同じにできます。

結果変数を、**COMMON**、**DATA**、整数 **POINTER**、または **EQUIVALENCE** ステートメントに指定することはできません。また、**PARAMETER**、**INTENT**、**OPTIONAL**、**SAVE**、または **VOLATILE** 属性を持つこともできません。結果変数が割り振り可能なオブジェクト、配列、ポインターではなく、かつ文字タイプでも派生型でもない場合にのみ、**STATIC** および **AUTOMATIC** 属性を指定することができます。

RESULT キーワードを指定する場合、**ENTRY** ステートメントは関数サブプログラムの中に置かなければなりません。また、*entry_name* は関数サブプログラムの有効範囲内の仕様ステートメントに置くことも、*result_name* を *entry_name* と同じにすることもできません。

結果変数をタイプ宣言ステートメントで初期化することはできません。

外部サブプログラムの中のエントリー名はグローバル・エンティティーです。モジュール・サブプログラムの中のエントリー名はグローバル・エンティティーではありません。インターフェース本体の中のプロシージャー名としてエントリー名を使用する場合にだけ、エントリー用のインターフェースをインターフェース・ブロックに置くことができます。

関数サブプログラムでは、*entry_name* は関数として呼び出しプロシージャーから参照される外部またはモジュール関数を示します。サブルーチン・サブプログラムでは、*entry_name* はサブルーチンなので、呼び出しプロシージャーからサブルーチンとして参照することができます。参照されると、**ENTRY** ステートメントに続く最初の実行可能ステートメントから実行が開始されます。

関数がエントリーから呼び出された場合、関数からの出る前に、結果変数を定義しなければなりません。

dummy_argument_list 内の名前を以下の場所に指定することはできません。

- **ENTRY** ステートメントの前にある実行可能ステートメントの中。ただし、その実行可能ステートメントより前にある **FUNCTION**、 **SUBROUTINE**、または **ENTRY** ステートメントにその名前が指定されている場合を除きます。
- ステートメント関数ステートメントの式の中。ただし、その名前がステートメント関数の仮引き数であり、**FUNCTION** または **SUBROUTINE** ステートメントに指定されているか、またはステートメント関数ステートメントより前にある **ENTRY** ステートメントに指定されている場合を除きます。

仮引き数の順序パラメーター、数字パラメーター、タイプ・パラメーター、kind タイプ・パラメーターは、**FUNCTION**、**SUBROUTINE**、または他の **ENTRY** ステートメントのものと異なっていてもかまいません。

オブジェクトの配列境界または文字長を指定するために宣言式の中で仮引き数を使用する場合、参照されるプロシージャー名の仮引き数リスト内に仮引き数があり、それが存在するときにのみ、プロシージャー参照中に実行されるステートメント内のオブジェクトを指定することができます。

再帰

ENTRY ステートメント自身を、直接的に参照することができるのは、サブプログラム・ステートメントが **RECURSIVE** を指定し、 **ENTRY** ステートメントが **RESULT** を指定している場合だけです。このようにすると、エントリー・プロシージャーは、サブプログラム内に明示インターフェースを持ちます。 **RESULT** 文節は、自分を間接的に参照するエントリーの場合は必要ありません。

Fortran 95

エレメント型サブプログラムには **ENTRY** ステートメントを指定することができますが、 **ENTRY** ステートメントに **ELEMENTAL** プレフィックスを指定することはできません。 **ELEMENTAL** プレフィックスを **SUBROUTINE** または **FUNCTION** ステートメント中に指定すると、 **ENTRY** ステートメントで定義されるプロシージャーはエレメント型になります。

Fortran 95 の終り

entry_name が文字タイプであり、関数が再帰的な場合、その長さにアスタリスクを指定することはできません。

IBM 拡張

-qrecur コンパイラー・オプションを指定すると、外部プロシージャーを再帰的に呼び出すことができます。しかし、プロシージャーが **RECURSIVE** または **RESULT** キーワードを指定する場合、XL Fortran は、このオプションを無視します。

IBM 拡張 の終り

例

```
RECURSIVE FUNCTION FNC() RESULT (RES)
   :
  ENTRY ENT () RESULT (RES)          ! The result variable name can be
                                     ! the same as for the function
   :
END FUNCTION
```

関連情報

- 378 ページの『FUNCTION』
- 473 ページの『SUBROUTINE』
- 209 ページの『再帰』
- 195 ページの『仮引き数』
- 「ユーザーズ・ガイド」の『**-qrecur** オプション』

EQUIVALENCE

目的

EQUIVALENCE ステートメントは 1 つの有効範囲単位内で複数のオブジェクトが同一のストレージを共用するように指定します。

構文

```
                 ┌─,──────────────────────────────────────┐
                 ▼                                        │
►►──EQUIVALENCE────(──equiv_object──,──equiv_object_list──)─┴──────────────►◄
```

equiv_object
: 変数名、配列エレメント、またはサブストリングです。どの添え字またはサブストリング式も、整数初期化式でなければなりません。

規則

equiv_object は、ターゲット、ポインター、仮引き数、関数名、ポインティング先、エントリー名、結果名、構造体コンポーネント、名前付き定数、自動データ・オブジェクト、割り振り可能オブジェクト、最終コンポーネントとして割り振り可能オブジェクトを含む派生型のオブジェクト、非順序派生型のオブジェクト、構造にポインターを含む順序派生型のオブジェクト、またはこれらのサブオブジェクトであってはなりません。

一対の括弧内で指定されたすべての項目は、始点が同じであるストレージ単位を占めるため、それらの項目は関連付けられます。これを*等価関連付け*と呼びます。等価関連付けによって他の項目も同様に関連付けられることがあります。

ストレージに関連した記憶単位のデフォルトの初期化を指定することができます。ただし、デフォルトの初期化を提供するデフォルトのオブジェクトまたはサブオブジェクトは、同じタイプでなければなりません。そのオブジェクトまたはサブオブジェクトは、同じタイプ・パラメーターでなければならず、記憶単位に同じ値を提供しなければなりません。

EQUIVALENCE ステートメントに配列エレメントを指定する場合、配列の次元の個数を超えて添え字の個数を指定することはできません。多次元配列を指定するのに単一の添え字 *n* を持つ配列エレメントを使用すると、配列のストレージ順序内の *n* エレメントが指定されます。それ以外の場合、 XL Fortran は、脱落している添え字を配列の対応する次元の下限値で置き換えます。添え字がなくサイズが 0 でない配列は、配列の最初のエレメントを参照します。

equiv_object が派生型の場合、それは順序派生型でなければなりません。

IBM 拡張

EQUIVALENCE ステートメント内でオブジェクトを使用できる場合、順序派生型のオブジェクトは順序派生型または組み込みデータ型のその他のオブジェクトと同等と見なすことができます。

XL Fortran では、関連する項目は組み込みタイプであっても順序派生型であってもかまいません。項目のデータ型が異なっていても、**EQUIVALENCE** ステートメントによってタイプの変換が起こることはありません。

IBM 拡張 の終り

関連する項目の長さが異なっていてもかまいません。

サイズが 0 であるすべての項目は、サイズが 0 ではない順序列の最初の文字ストレージ単位と同じストレージを共用します。

EQUIVALENCE ステートメントを使っても、2 つの異なる共通ブロックのストレージ順序が関連させられることはありません。また、1 つのストレージ順序内に同一のストレージ単位が 2 回以上現れるように指定することはできません。 **EQUIVALENCE** ステートメントは、それ自身や、前に **EQUIVALENCE** ステートメントで設定された関連付けと矛盾してはいけません。

EQUIVALENCE ステートメントを使用して、共通ブロック内にない名前と共通ブロック内の名前との間でストレージを共用させることができます。

EQUIVALENCE ステートメントによって宣言されたオブジェクトが **PROTECTED** 属性を持つように指定した場合、その **EQUIVALENCE** ステートメントで指定されたオブジェクトはすべて **PROTECTED** 属性を持たなければなりません。

EQUIVALENCE ステートメントを使用して共通ブロックを拡張できますが、拡張部分は最後のエントリーの後に追加できるだけで、最初のエントリーの前には追加できません。たとえば、**EQUIVALENCE** ステートメントを使用して、共通ブロック内の変数に関連させようとする変数が配列エレメントの場合、その配列の他のエレメントの暗黙の関連付けによって、共通ブロックの大きさが拡張する場合があります。

例

```
DOUBLE PRECISION A(3)
REAL B(5)
EQUIVALENCE (A,B(3))
```

ストレージ単位の関連付け:

```
           |      |      |      |      |      |      |      |
Array A:                 |     A(1)    |     A(2)    |     A(3)    |
Array B:   | B(1) | B(2) | B(3) | B(4) | B(5) |
```

次の例は、2 つの項目の関連付けの結果、別の関連付けが行われることを示しています。

```
AUTOMATIC A
CHARACTER A*4,B*4,C(2)*3
EQUIVALENCE (A,C(1)),(B,C(2))
```

ストレージ単位の関連付け:

```
              |      |      |      |      |      |      |      |
Variable A:   |             A             |
Variable B:                        |             B             |
Array C:      |        C(1)        |        C(2)        |
```

XL Fortran は A と B の両方を C クラスに関連させるので、A と B は互いに関連し合うことになります。これらはすべて、自動ストレージ・クラスを持ちます。

```
INTEGER(4)   G(2,-1:2,-3:2)
REAL(4)      H(3,1:3,2:3)
EQUIVALENCE  (G(2),H(1,1))   ! G(2) is G(2,-1,-3)
                             ! H(1,1) is H(1,1,2)
```

関連情報

- 78 ページの『変数のストレージ・クラス』
- 70 ページの『変数の定義状況』

EXIT

目的

EXIT ステートメントは、 **DO** 構造体または **DO WHILE** 構造体が繰り返しをすべて完了する前に、その構造体の実行を終了させます。

構文

```
►►──EXIT──┬──────────────────┬─────────────────────────────────────────────►◄
          └─DO_construct_name─┘
```

DO_construct_name
: **DO** または **DO WHILE** 構造体の名前です。

規則

EXIT ステートメントは、**DO** 構造体または **DO WHILE** 構造体の中に置かれ、*DO_construct_name* によって指定された **DO** または **DO WHILE** 構造体に所属します。 *DO_construct_name* が指定されていない場合は、そのステートメントを直接包含する **DO** または **DO WHILE** 構造体に所属します。 *DO_construct_name* を指定するときは、**EXIT** ステートメントはその構造体の範囲内になければばなりません。

EXIT ステートメントを実行すると、**EXIT** ステートメントが属する **DO** または **DO WHILE** 構造体は非アクティブになります。 **EXIT** ステートメントが他の **DO** または **DO WHILE** 構造体にネストされている場合、それらも非アクティブになります。 **DO** 変数は最後に定義した値を保持します。 **DO** 構造体は、構造体制御がない場合、非アクティブになるまで無制限に繰り返しを実行します。 **EXIT** ステートメントを使用して、構造体を非アクティブにすることができます。

EXIT ステートメントには、ステートメント・ラベルを付けることができます。しかし、**DO** または **DO WHILE** 構造体を終了させるラベル付きステートメントとして使用することはできません。

例

```
      LOOP1: DO I = 1, 20
         N = N + 1
10       IF (N > NMAX) EXIT LOOP1          ! EXIT from LOOP1

         LOOP2: DO WHILE (K==1)
            KMAX = KMAX - 1
20          IF (K > KMAX) EXIT             ! EXIT from LOOP2
         END DO LOOP2
```

```
          LOOP3:  DO J = 1, 10
                     N = N + 1
30                   IF (N > NMAX) EXIT LOOP1       ! EXIT from LOOP1
                     EXIT LOOP3                     ! EXIT from LOOP3
                  END DO LOOP3

        END DO LOOP1
```

関連情報

- 152 ページの『DO 構造体』
- 157 ページの『DO WHILE 構造体』

EXTERNAL

目的

EXTERNAL 属性は、名前を外部プロシージャー、仮プロシージャー、ブロック・データ・プログラム単位として指定します。 **EXTERNAL** 属性を持つプロシージャー名は実引き数として使用できます。

構文

```
►►──EXTERNAL──┬──────┬──name_list──────────────────────────────►◄
              └─ :: ─┘
```

name 外部プロシージャー、仮プロシージャー、または **BLOCK DATA** プログラム単位の名前です。

規則

外部プロシージャー名または仮引き数名を実引き数として使用する場合は、**EXTERNAL** 属性で指定するか、または、その有効範囲単位内のインターフェース・ブロックで宣言しなければなりません。ただし、それらの両方で指定することはできません。

1 つの有効範囲単位内で **EXTERNAL** 属性を持たせて組み込みプロシージャー名を指定すると、その名前はユーザー定義の外部プロシージャー名になります。したがって、それと同じ名前の組み込みプロシージャーを、その有効範囲単位から呼び出すことはできません。

1 つの有効範囲単位内では、1 つの名前に **EXTERNAL** 属性は 1 回しか指定できません。

有効範囲単位内のインターフェース・ブロックでは、**EXTERNAL** ステートメントに指定した名前を、特定のプロシージャー名として指定することはできません。

EXTERNAL 属性と互換性のある属性

- OPTIONAL
- PRIVATE
- PUBLIC

例

```
PROGRAM MAIN
  EXTERNAL AAA
  CALL SUB(AAA)          ! Procedure AAA is passed to SUB
END

SUBROUTINE SUB(ARG)
  CALL ARG()             ! This results in a call to AAA
END SUBROUTINE
```

関連情報

- 205 ページの『仮引き数としてのプロシージャー』
- 923 ページの『付録 A. 異なる標準の間の互換性』の項目 4

FORALL

Fortran 95

目的

FORALL ステートメントは、サブオブジェクトのグループへの割り当て、特に配列エレメントへの割り当てを実行します。 **WHERE** ステートメントとは異なり、割り当ては配列レベルではなく、要素レベルで実行されます。 **FORALL** ステートメントでは、ポインター割り当ても可能です。

構文

```
▶▶──FORALL──forall_header──forall_assignment──────────────────────────▶◀
```

forall_header

```
▶▶──(──forall_triplet_spec_list──┬──────────────────────┬──)──────────▶◀
                                 └──,──scalar_mask_expr──┘
```

forall_triplet_spec

```
►►─index_name─ = ─subscript─ : ─subscript─ : ─subscript─┬────────────┬─►◄
                                                         └─ : ─stride─┘
```

forall_assignment
  assignment_statement または *pointer_assignment_statement* のどちらかです。

scalar_mask_expr
  スカラー論理式です。

subscript、*stride*
  スカラー整数式です。

規則

forall_header のマスク式、および *forall_assignment* の中で参照できるのは、純粋プロシージャーだけです (定義済み操作または割り当てによって参照されるものを含む)。

index_name は、スカラー整数の変数でなければなりません。また、これはステートメント・エンティティーです。つまり、有効範囲単位内の他のエンティティーに影響を与えたり、影響を受けたりしません。

forall_triplet_spec_list の中の *subscript* および *stride* には、*forall_triplet_spec_list* の中の *index_name* への参照を含めることはできません。 *forall_header* の中の式の評価は、*forall_header* の中の他の式の評価に影響を与えてはなりません。

forall_triplet_spec が次のとおりであるとします。

```
index1 = s1:s2:s3
```

指標の最大値は、次のようにして判別されます。

```
max = INT((s2-s1+s3)/s3)
```

ストライド (上記の *s3*) が指定されない場合、値 1 が想定されます。どの指標でも *max* ≤ 0 の場合、 *forall_assignment* は実行されません。以下に例を示します。

```
index1 = 2:10:3    !  The index values are 2,5,8.
                      max = INT((10-2+3)/3) = 3.

index2 = 6:2:-1    !  The index values are 6,5,4,3,2.
index2 = 6:2       !  No index values.
```

マスク式が省略されると、.TRUE. の値が想定されます。

アトミック・オブジェクトを複数回割り当てることはできません。アトミックでないオブジェクトへの割り当ては、すべてのサブオブジェクトへ割り当てるか、またはターゲットをすべてのサブオブジェクトと関連させます。

FORALL ステートメントの解釈

1. それぞれの *forall_triplet_spec* について、順序に関係なく、*subscript* 式および *stride* 式を評価します。可能な *index_name* 値のすべての組が、組み合わせの集合を形成します。たとえば、次のようなステートメントを考えます。

   ```
   FORALL (I=1:3,J=4:5) A(I,J) = A(J,I)
   ```

   `I` および `J` の組み合わせの集合は次のとおりです。

   ```
   {(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)}
   ```

   -1 および **-qnozerosize** コンパイラー・オプションは、このステップに影響を与えません。

2. それぞれの組み合わせについて順序に関係なく *scalar_mask_expr* を評価し、アクティブな組み合わせの集合 (*scalar_mask_expr* で `.TRUE.` と評価されたもの) を作ります。たとえば、(`I+J.NE.6`) が上記の集合に適用された場合、アクティブな組み合わせの集合は次のようになります。

   ```
   {(1,4),(2,5),(3,4),(3,5)}
   ```

3. *assignment_statement* の場合、すべてのアクティブな *index_name* 値の組み合わせについて、順序に関係なく、右方サイド *expression* 内のすべての値、および左方サイド *variable* 内のすべての添え字、ストライド、およびサブストリング境界を評価します。

   pointer_assignment の場合、すべてのアクティブな *index_name* 値の組み合わせについて、順序に関係なく、何がポインター割り当てのターゲットになるのかを判別し、すべての添え字、ストライド、およびサブストリング境界を評価します。ターゲットがポインターであるかどうかにかかわりなく、ターゲットの判別には、その値の評価は含まれません。

4. *assignment_statement* の場合、すべてのアクティブな *index_name* 値の組み合わせについて、順序に関係なく、計算された *expression* の値を、対応する *variable* エンティティーに割り当てます。

   pointer_assignment の場合、すべてのアクティブな *index_name* 値の組み合わせについて、順序に関係なく、すべてのターゲットを、対応するポインター・エンティティーに関連させます。

ループの並列化

FORALL ステートメントおよび **FORALL** 構造体は、代入ステートメントの並列化が可能となるように設計されています。 **FORALL** 内で代入ステートメントを実行する場合、オブジェクトの割り当ては、他のオブジェクトの割り当てに影響を与えません。次の例では、結果を変更しないで、順序に関係なく、A のエレメントへの割り当てを実行できます。

```
FORALL (I=1:3,J=1:3) A(I,J)=A(J,I)
```

IBM 拡張

INDEPENDENT ディレクティブを指定すると、 **DO** ループの各反復または **FORALL** ステートメントあるいは **FORALL** 構造体内での各操作を、任意の順序で、プログラムのセマンティクスに影響を与えずに実行することができます。**FORALL** ステートメントまたは **FORALL** 構造体内の操作は、以下のように定義されます。

- *mask* の評価
- 右側と左側の指標の両方またはどちらか一方の評価
- 割り当ての評価

したがって、以下のループ

```
         INTEGER, DIMENSION(2000) :: A,B,C
!IBM*  INDEPENDENT
         DO I = 1, 1999, 2
           A(I) = A(I+1)
         END DO
```

は、意味上では以下の配列割り当てに等価です。

```
         INTEGER, DIMENSION(2000) :: A,B,C
         A(1:1999:2) = A(2:2000:2)
```

ヒント

特定の **FORALL** を並列処理することが可能であり、利点がある場合には、**FORALL** の前に、**INDEPENDENT** ディレクティブを指定してください。 XL Fortran は、 **FORALL** を並列化することが有効かどうかを必ず判別できるわけではないので、 **INDEPENDENT** ディレクティブによって有効であることが保証されます。

IBM 拡張 の終り

例

```
INTEGER A(1000,1000), B(200)
I=17
FORALL (I=1:1000,J=1:1000,I.NE.J) A(I,J)=A(J,I)
PRINT *, I    ! The value 17 is printed because the I
              ! in the FORALL has statement scope.
FORALL (N=1:200:2) B(N)=B(N+1)
END
```

関連情報

- 127 ページの『組み込み割り当て』
- 142 ページの『ポインターの割り当て』

- 139 ページの『FORALL 構造体』
- 519 ページの『INDEPENDENT』
- 163 ページの『ステートメント・エンティティーおよび構造体エンティティー』

Fortran 95 の終り

FORALL (構造体)

Fortran 95

目的

FORALL (構造体) ステートメントは、 **FORALL** 構造体の最初のステートメントです。

構文

```
>>-+-----------------------------+--FORALL--+---------------+----------><
   '-FORALL_construct_name-- : --'          '-forall_header-'
```

forall_header

```
>>-(--forall_triplet_spec_list--+-----------------------+--)-----------><
                                '-,--scalar_mask_expr---'
```

forall_triplet_spec

```
>>-index_name-- = --subscript-- : --subscript--+--------------+--------><
                                               '- : --stride--'
```

scalar_mask_expr
: スカラー論理式です。

subscript、*stride*
: 両方ともスカラー整数式です。

規則

forall_header のマスク式で参照されるプロシージャーはすべて、(定義済みの操作または割り当てによって参照されるものを含む) 純粋でなければなりません。

index_name は、スカラー整数の変数でなければなりません。 *index_name* の有効範囲は、**FORALL** 構造体の全体です。

forall_triplet_spec_list の中の *subscript* および *stride* には、*forall_triplet_spec_list* の中の *index_name* への参照を含めることはできません。 *forall_header* の中の式の評価は、*forall_header* の中の他の式の評価に影響を与えてはなりません。

forall_triplet_spec が次のとおりであるとします。

```
index1 = s1:s2:s3
```

指標の最大値は、次のようにして判別されます。

```
max = INT((s2-s1+s3)/s3)
```

ストライド (上記の *s3*) が指定されない場合、値 1 が想定されます。どの指標でも *max* ≤ 0 の場合、 *forall_assignment* は実行されません。たとえば、次のようになります。

```
index1 = 2:10:3    !  The index values are 2,5,8.
                   !  max = floor(((10-2)/3)+1) = 3.

index2 = 6:2:-1    !  The index values are 6,5,4,3,2.
index2 = 6:2       !  No index values.
```

マスク式が省略されると、.TRUE. の値が想定されます。

例

```
POSITIVE: FORALL (X=1:100,A(X)>0)
  I(X)=I(X)+J(X)
  J(X)=J(X)-I(X+1)
END FORALL POSITIVE
```

関連情報

- 351 ページの『END (構造体)』
- 139 ページの『FORALL 構造体』
- 163 ページの『ステートメント・エンティティーおよび構造体エンティティー』

Fortran 95 の終り

FORMAT

目的

FORMAT ステートメントは I/O ステートメントに形式仕様を提供します。

構文

```
►►──FORMAT──(──┬──────────────────┬──)──────────────────────────────────────►◄
               └─format_item_list─┘
```

format_item

```
►►──┬──┬───┬──data_edit_desc──────────┬─────────────────────────────────────►◄
    │  └─r─┘                          │
    ├─control_edit_desc───────────────┤
    ├──┬───┬──(──format_item_list──)──┤
    │  └─r─┘                          │
    └─char_string_edit_desc───────────┘
```

r
: 無符号の正の整リテラル定数であり、kind 型付きパラメーターは指定できません。または、整数値への計算を行う算術式をかぎ括弧 (< と >) で囲んで指定します。これは繰り返し仕様と呼ばれます。*format_item_list* または *data_edit_desc* を繰り返す回数を指定します。デフォルトは 1 です。

data_edit_desc
: データ編集記述子です。

control_edit_desc
: 制御編集記述子です。

char_string_edit_desc
: 文字ストリング編集記述子です。

データ編集記述子

形式	使途	ページ
A **A***w*	文字値を編集します。	244
B*w* **B***w.m*	2 進値を編集します。	245
E*w.d* **E***w.d***E***e* **E***w.d***D***e* * **E***w.d***Q***e* * **D***w.d* **EN***w.d* **EN***w.d***E***e* **ES***w.d* **ES***w.d***E***e* **Q***w.d* *	指数付き実数および複素数を編集します。	247

形式	使途	ページ
F*w.d*	指数なしの実数および複素数を編集します。	251
G*w.d* **G***w.d***E***e* **G***w.d***D***e* * **G***w.d***Q***e* *	データのタイプに出力形式を適用し、組み込みタイプのデータ・フィールドを編集ます。また、データのタイプが実数の場合、データの絶対値を編集します。	253
I*w* **I***w.m*	整数を編集します。	255
L*w*	論理値を編集します。	256
O*w* **O***w.m*	8 進値を編集します。	257
Q *	入力レコード内に残っている文字のカウントを戻します。 *	259
Z*w* **Z***w.m*	16 進値を編集します。	260

注: * IBM 拡張

それぞれの意味は次のとおりです。

w すべてのブランクを含む、フィールドの幅を指定します。これは正でなければなりません。 F95 ただし、出力で、**I**、**B**、**O**、**Z**、および **F** 編集記述子でこれにゼロを指定できる、 Fortran 95 は例外です。 F95

m 印刷する桁数を示します。

d 小数点以下の桁数を指定します。

e 指数フィールド内の桁数を指定します。

w、*m*、*d*、*e* は、以下のように表すこともできます。
- 無符号のリテラル整数

IBM 拡張

- 不等号括弧 (**<** と **>**) で囲まれているスカラー整数式。 詳細については、 377 ページの『変数形式設定式』を参照してください。

IBM 拡張 の終り

w、*m*、*d*、または *e* に対して kind パラメーターを指定することはできません。

IBM 拡張

注:

Q データ編集記述子には 2 つのタイプがあります (**Q***w.d* および **Q**)。

拡張精度 Q
Q*w.d* という構文からなる **Q** 編集記述子です。

文字カウント Q
Q という構文からなる **Q** 編集記述子です。

IBM 拡張 の終り

制御編集記述子

形式	使途	ページ
/ *r* /	現在のレコードに関するデータ転送の終わりを指定します。	262
:	I/O リスト内にこれ以上項目がない場合に、形式制御の終わりを指定します。	262
$ *	出力でレコードの終わりを抑制します。*****	263 *****
BN	数値入力フィールド内の非先行ブランクを無視します。	264
BZ	数値入力フィールド内の非先行ブランクをゼロとして解釈します。	264
*k***P**	実数および複素数項目に対してスケール因数を指定します。	266
S SS	正符号を書き込まないように指定します。	267
SP	正符号を書き込むように指定します。	267
T*c*	次の文字の転送先または転送元のレコード内での絶対位置を指定します。	268
TL*c*	次の文字の転送先または転送元のレコード内での相対位置 (レコード内の現在の位置の左側の位置) を指定します。	268
TR*c*	次の文字の転送先または転送元のレコード内での相対位置 (レコード内の現在の位置の右側の位置) を指定します。	268

形式	使途	ページ
*o***X**	次の文字の転送先または転送元のレコード内での相対位置 (レコード内の現在の位置の右側の位置) を指定します。	268

注: * IBM 拡張

それぞれの意味は次のとおりです。

r　繰り返し指定子です。これは、無符号かつ正のリテラル整定数です。

k　使用するスケール因数を指定します。これは、オプションの符号付きリテラル整定数です。

c　レコード内の文字位置を指定します。これは、無符号かつゼロ以外のリテラル整定数です。

o　レコード内の相対文字位置です。これは、無符号かつゼロ以外のリテラル整定数です。

IBM 拡張

r、*k*、*c*、および *o* は、整数値に計算される不等号括弧 (**<** と **>**) で囲まれた算術式としても表すことができます。

IBM 拡張 の終り

r、*k*、*c*、*o* に対して kind 型付きパラメーターを指定することはできません。

文字ストリング編集記述子

形式	使途	ページ
*n***H***str*	文字ストリング (str) を出力します。	265
'*str*' "*str*"	文字ストリング (*str*) を出力します。	263

n　リテラル・フィールド内の文字数です。これは、無符号かつ正のリテラル整定数です。ブランクは文字のカウントに含まれます。 kind 型付きパラメーターを指定することはできません。

規則

定様式の **READ**、**WRITE**、または **PRINT** ステートメント内の形式識別子が、ステートメント・ラベルであるか、あるいはステートメント・ラベルが割り当てられた変数である場合、ステートメント・ラベルは **FORMAT** ステートメントを識別します。

FORMAT ステートメントには、ステートメント・ラベルを付けなければなりません。**FORMAT** ステートメントを、 ブロック・データ・プログラム単位、インターフェース・ブロック、モジュールの有効範囲、または派生型定義に指定することはできません。

コンマは編集記述子を分離します。 **P** 編集記述子と、その直後に続く **F**、**E**、 **EN**、**ES**、**D**、**G**、または **Q** (拡張精度と文字カウントの両方) 編集記述子との間のコンマ、オプションの繰り返し指定がない場合の、スラッシュ編集記述子の前のコンマ、スラッシュ編集記述子の後のコンマ、およびコロン編集記述子の前後のコンマは省略できます。

FORMAT 仕様を I/O ステートメントの中で文字式として指定することもできます。

XL Fortran は形式仕様の中で大文字と小文字を同じものとして取り扱います。ただし、文字ストリング編集記述子の場合はその限りではありません。

文字形式仕様

定様式の **READ**、**WRITE**、または **PRINT** ステートメント内の形式識別子 (447ページ) が文字配列名または文字式の場合、その配列または式の値が文字配列仕様です。

形式識別子が文字配列エレメント名の場合、形式仕様は、その配列エレメントの中に完全に入っていなければなりません。形式識別子が文字配列名の場合、形式仕様は、最初のエレメントから後続のエレメントにまたがっていてもかまいません。

形式仕様の前にブランクがあってもかまいません。形式仕様の終了を示す右括弧の次に文字データを指定してもかまいません。そのようにしても、形式仕様には何の影響もありません。

変数形式設定式:

IBM 拡張

編集記述子が整定数を必要とするときは、いつでも **FORMAT** ステートメントに整数式を指定することができます。整数式はかぎ括弧 (**<** と **>**) で囲まなくてはなりません。変数形式設定式の外側に符号を使用することはできません。 次に、使用可能な形式仕様を示します。

```
      WRITE(6,20) INT1
20    FORMAT(I<MAX(20,5)>)

      WRITE(6,FMT=30) INT2, INT3
30    FORMAT(I<J+K>,I<2*M>)
```

整数式は、有効であれば関数呼び出しおよび仮引き数への参照を含むどんな Fortran 式であってもかまいません。ただし、次の制約があります。

- 式を **H** 編集記述子と併用することはできません。

- 式にはグラフィック関係の演算子を入れることはできません。

READ、**WRITE**、または **PRINT** ステートメントの実行中に I/O 項目を処理するたびに、式の値は再計算されます。

IBM 拡張 の終り

例

```
      CHARACTER*32 CHARVAR
      CHARVAR="('integer: ',I2,'  binary: ',B8)"  ! Character format
      M = 56                                      ! specification
      J = 1                                       !     OUTPUT:
      X = 2355.95843                              !
      WRITE (6,770) M,X                           !  56   2355.96
      WRITE (6,CHARVAR) M,M                       ! integer: 56
                                                  ! binary: 00111000
      WRITE (6,880) J,M                           !  1
                                                  ! 56
770   FORMAT(I3, 2F10.2)
880   FORMAT(I<J+1>)
      END
```

関連情報

- 239 ページの『第 9 章 I/O の形式設定』
- 436 ページの『PRINT』
- 445 ページの『READ』
- 500 ページの『WRITE』

FUNCTION

目的

FUNCTION ステートメントは、関数サブプログラムの最初のステートメントです。

構文

```
   ┌──────────┐
   ▼          │
►►─┬──────────┴─FUNCTION─name─┬──────────────────┬──────────────────────►
   └─prefix─┘                 │   (1)       (2)  │
                              └─*───────len──────┘

►─(─┬─────────────────────┬─)─┬──────────────────────────────┬──────────►◄
    └─dummy_argument_list─┘   └─RESULT─(─result_name─)─┘
```

注:

1 IBM 拡張

2 IBM 拡張

prefix 次のうちの 1 つです。

type_spec

RECURSIVE

F95 **PURE** F95

F95 **ELEMENTAL** F95

type_spec

関数結果のタイプと型付きパラメーターを指定します。 *type_spec* の詳細については、 481 ページの『タイプ宣言』を参照してください。

name 関数サブプログラムの名前です。

IBM 拡張

len 符号なし整数リテラル、または括弧で囲んだスカラー整数初期化式のいずれかです。この値は関数の結果変数の長さを指定します。 **FUNCTION** ステートメントでタイプを指定した場合のみ、この値を指定できます。 タイプは **DOUBLE PRECISION**、**DOUBLE COMPLEX**、**BYTE** または派生型であってはなりません。

IBM 拡張 の終り

規則

最大 1 つの *prefix* を指定できます。

関数結果のタイプと型付きパラメーターは *type_spec* でも関数サブプログラムの宣言部分で結果変数を宣言することによっても指定できます。ただし、両用の方法を同時に使用することはできません。指定しない場合は、暗黙の入力規則が適用されます。長さ指定子は *type_spec* と *len* の両方では指定できません。

RESULT を指定すると、*result_name* は関数結果変数になります。 *name* は、サブプログラム内のどの仕様ステートメントにおいても宣言することはできませんが、参照することはできます。 *result_name* を *name* と同じにすることはできません。 **RESULT** を指定しないと、*name* は関数結果変数になります。

結果変数が配列またはポインターの場合、**DIMENSION** または **POINTER** 属性はそれぞれ関数本体で指定しなければなりません。

関数結果がポインターの場合、結果変数の形状によって関数が戻す値の形状を決定します。結果変数がポインターの場合、関数はターゲットをポインターと関連させるか、あるいはポインターの関連付け状況を非関連として定義しなければなりません。

結果変数がポインターでない場合、関数はその値を定義しなければなりません。

外部関数名が派生型であり、タイプが使用関連付けまたはホスト関連付けでない場合、その派生型は順序派生型でなければなりません。

関数結果変数を変数形式設定式の中に置いてはいけません。また、関数結果変数は、**COMMON**、**DATA**、整数 **POINTER**、または **EQUIVALENCE** ステートメントで指定することも、 **PARAMETER**、**INTENT**、**OPTIONAL**、または **SAVE** 属性を持つこともできません。結果変数が割り振り可能なオブジェクト、配列、またはポインターではなく、かつ文字タイプでも派生型でもない場合にのみ、**STATIC** および **AUTOMATIC** 属性を指定することができます。

関数結果変数は入り口プロシージャーの結果変数と関連を持ちます。このことを入り口関連付けといいます。これらの結果変数のうちの 1 つを定義すると、関連している変数のうち、同じタイプのすべての変数が定義され、どのエントリー・ポイントから入ったかに関係なく関数の値になります。

関数サブプログラムに入り口プロシージャーが含まれ、タイプが文字タイプでも派生型でもなく、結果変数が **POINTER** 属性を持つか、またはスカラーではない場合、結果変数は同じタイプである必要はありません。サブプログラムの中で **RETURN** または **END** ステートメントを実行するときに、関数の参照用に名前が使用される変数は定義済み状態になっていなければなりません。関連した変数でタイプが異なるものは、関数の参照中に定義済み状態にすることはできません。ただし、関連した同じタイプの変数の 1 つによってサブプログラムの実行中に、後から再定義する場合は除きます。

再帰

以下の場合、キーワード **RECURSIVE** を直接または間接的に指定しなければなりません。
- 関数がそれ自身を呼び出す
- 関数が同じサブプログラムの **ENTRY** ステートメントによって定義される関数を呼び出す
- 同じサブプログラム内の入り口プロシージャーがそれ自体を呼び出す

- 同じサブプログラム内の入り口プロシージャーが同じサブプログラム内の他の入り口プロシージャーを呼び出す
- 同じサブプログラム内の入り口プロシージャーが **FUNCTION** ステートメントによって定義されるサブプログラムを呼び出す

関数がそれ自身を直接呼び出すには、 **RECURSIVE** と **RESULT** の両方のキーワードを指定しなければなりません。この両方のキーワードが存在すると、サブプログラム内に明示プロシージャー・インターフェースが作成されます。

name が文字タイプであり、関数が再帰的な場合、その長さにアスタリスクを指定することはできません。

IBM 拡張

RECURSIVE を指定した場合、結果変数のデフォルトのストレージ・クラスは自動になります。

-qrecur コンパイラー・オプションを指定すると、外部プロシージャーを再帰的に呼び出すことができます。しかし、**FUNCTION** ステートメントが **RECURSIVE** または **RESULT** を指定する場合、XL Fortran はこのオプションを無視します。

IBM 拡張 の終り

エレメント型プロシージャー

Fortran 95

エレメント型プロシージャーでは、キーワード **ELEMENTAL** を指定しなければなりません。 **ELEMENTAL** キーワードを指定している場合、**RECURSIVE** キーワードを指定することはできません。

Fortran 95 の終り

例

```
RECURSIVE FUNCTION FACTORIAL (N) RESULT (RES)
  INTEGER RES
  IF (N.EQ.0) THEN
    RES=1
  ELSE
    RES=N*FACTORIAL(N-1)
  END IF
END FUNCTION FACTORIAL
```

```
PROGRAM P
  INTERFACE OPERATOR (.PERMUTATION.)
    ELEMENTAL FUNCTION MYPERMUTATION(ARR1,ARR2)
      INTEGER :: MYPERMUTATION
```

```
    INTEGER, INTENT(IN) :: ARR1,ARR2
  END FUNCTION MYPERMUTATION
END INTERFACE

INTEGER PERMVEC(100,150),N(100,150),K(100,150)
...
PERMVEC = N .PERMUTATION. K
...
END
```

関連情報

- 188 ページの『関数およびサブルーチン・サブプログラム』
- 359 ページの『ENTRY』
- 190 ページの『関数参照』
- 195 ページの『仮引き数』
- 468 ページの『ステートメント関数』
- 209 ページの『再帰』
- 「ユーザーズ・ガイド」の『**-qrecur** オプション』
- 209 ページの『純粋プロシージャー』
- 212 ページの『エレメント型プロシージャー』

GO TO (割り当て)

目的

割り当て **GO TO** ステートメントは、プログラム制御を実行可能ステートメントに移します。そのステートメント・ラベルは **ASSIGN** ステートメントの中で指定されます。

構文

```
>>—GO TO—variable_name—┬──────────────────────────────┬——————————————><
                       └─┬───┬─(—stmt_label_list—)─┘
                         └─,─┘
```

variable_name
: **INTEGER(4)** または **INTEGER(8)** タイプのスカラー変数名です。これは **ASSIGN** ステートメントの中でステートメント・ラベルが割り当てられています。

stmt_label
: 割り当て **GO TO** と同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルです。同じステートメント・ラベルを *stmt_label_list* 内に 2 回以上指定することができます。

規則

割り当て **GO TO** ステートメントを実行する場合、ステートメント・ラベルの値を使用して *variable_name* に指定する変数は定義済みでなければなりません。この定義を設定するには、割り当て **GO TO** ステートメントと同じ有効範囲単位内の **ASSIGN** ステートメントを使用する必要があります。整変数がサブプログラム内の仮引き数である場合、サブプログラムでその変数にステートメント・ラベルを割り当ててから、その変数を割り当て **GO TO** ステートメントで使用してください。割り当て **GO TO** ステートメントを実行すると、そのステートメント・ラベルで識別したステートメントに制御が移ります。

stmt_label_list を指定する場合には、*variable_name* で指定した変数に割り当てるステートメント・ラベルを、リスト内に指定しなければなりません。

割り当て **GO TO** を **DO** または **DO WHILE** 構造体の終端ステートメントにすることはできません。

Fortran 95

割り当て **GO TO** ステートメントは Fortran 95 では削除されています。

Fortran 95 の終り

例

```
      INTEGER RETURN_LABEL
   ⋮
!  Simulate a call to a local procedure
      ASSIGN 100 TO RETURN_LABEL
      GOTO 9000
100   CONTINUE
   ⋮
9000  CONTINUE
!  A "local" procedure
   ⋮
      GOTO RETURN_LABEL
```

関連情報

- 382 ページの『GO TO (割り当て)』
- 14 ページの『ステートメント・ラベル』
- 157 ページの『分岐』
- 927 ページの『削除された機能』

GO TO (計算)

目的

計算 **GO TO** ステートメントは、複数の実行可能ステートメントの 1 つにプログラム制御を移します。

構文

```
▶▶─GO TO─(─stmt_label_list─)─┬───┬─arith_expr─────────────────────▶◀
                             └─,─┘
```

stmt_label
: 計算 **GO TO** と同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルです。同じステートメント・ラベルを *stmt_label_list* 内に 2 回以上指定することができます。

arith_expr
: スカラー整数式です。

IBM 拡張

実数または複素数にすることもできます。 式の値が整数でない場合、使用前に XL Fortran によって **INTEGER(4)** に変換されます。

IBM 拡張 の終り

規則

計算 **GO TO** ステートメントを実行すると、*arith_expr* が計算されます。その結果の値が *stmt_label_list* への指標として使用されます。次に、制御が、その指標で識別されるステートメント・ラベルを持つステートメントに移ります。たとえば、*arith_expr* の値が 4 であれば、*stmt_label_list* 内の 4 番目にあるステートメント・ラベルを持つステートメントに制御が移ります。ただしこの場合、リストには少なくとも 4 つのラベルが存在していなければなりません。

arith_expr の値が 1 より小さいか、またはリスト内のステートメント・ラベルの個数より大きい場合、**GO TO** ステートメントは機能せず (**CONTINUE** ステートメントと同様に)、その次のステートメントが実行されます。

例

```
        INTEGER NEXT
    ⋮
        GO TO (100,200) NEXT
10      PRINT *,'Control transfers here if NEXT does not equal 1 or 2'
    ⋮
100     PRINT *,'Control transfers here if NEXT = 1'
    ⋮
200     PRINT *,'Control transfers here if NEXT = 2'
```

関連情報

- 14 ページの『ステートメント・ラベル』
- 157 ページの『分岐』

GO TO (無条件)

目的

無条件 **GO TO** ステートメントは、特定の実行可能ステートメントにプログラム制御を移します。

構文

```
►►──GO TO──stmt_label──────────────────────────────────────────────►◄
```

stmt_label
無条件 **GO TO** と同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルです。

規則

無条件 **GO TO** ステートメントは、*stmt_label* によって識別されるステートメントに制御を移します。

無条件 **GO TO** ステートメントを **DO** または **DO WHILE** 構造体の終端ステートメントとして指定することはできません。

例

```
   REAL(8) :: X,Y
   GO TO 10
   ⋮
10 PRINT *, X,Y
   END
```

関連情報

- 14 ページの『ステートメント・ラベル』
- 157 ページの『分岐』

IF (算術)

目的

算術 **IF** ステートメントは、算術式の計算に従って、3 つの実行可能ステートメントのうちの 1 つにプログラム制御を移します。

構文

```
►►—IF—(—arith_expr—)—stmt_label1—,—stmt_label2—,—stmt_label3————————►◄
```

arith_expr
: 整数または実数タイプの算術式です。

stmt_label1、*stmt_label2*、および *stmt_label3*
: **IF** ステートメントと同じ有効範囲単位内にある実行可能ステートメントのステートメント・ラベルです。これら 3 つのステートメント・ラベルの中には、同じステートメント・ラベルが 2 つ以上あってもかまいません。

規則

算術 **IF** ステートメントを実行すると、*arith_expr* が計算され、*arith_expr* の値がゼロより小さいか、ゼロか、ゼロより大きいかによって、それぞれ *stmt_label1*、 *stmt_label2*、*stmt_label3* で識別されるステートメントに制御が移されます。

例

```
      IF (K-100) 10,20,30
10    PRINT *,'K is less than 100.'
      GO TO 40
20    PRINT *,'K equals 100.'
      GO TO 40
30    PRINT *,'K is greater than 100.'
40    CONTINUE
```

関連情報

- 157 ページの『分岐』
- 14 ページの『ステートメント・ラベル』

IF (ブロック)

目的

ブロック **IF** ステートメントは、**IF** 構造体内の最初のステートメントです。

構文

```
>>─┬─────────────────────┬─IF─(─scalar_logical_expr─)─THEN──────────────><
   └─IF_construct_name─:─┘
```

IF_construct_name
: **IF** 構造体を識別する名前です。

規則

ブロック **IF** ステートメントは論理式を計算し、**IF** 構造体に含まれるブロックのうちの最大 1 つを実行します。

IF_construct_name を指定する場合、この名前を **END IF** ステートメントで必ず指定する必要がありますが、**IF** 構造体内の **ELSE IF** または **ELSE** ステートメントへの指定は任意です。

例

```
WHICHC: IF (CMD .EQ. 'RETRY') THEN
     IF (LIMIT .GT. FIVE) THEN          ! Nested IF constructs
   ⋮
         CALL STOP
     ELSE
         CALL RETRY
     END IF
ELSE IF (CMD .EQ. 'STOP') THEN WHICHC
     CALL STOP
ELSE IF (CMD .EQ. 'ABORT') THEN
     CALL ABORT
ELSE WHICHC
     GO TO 100
END IF WHICHC
```

関連情報

- 147 ページの『IF 構造体』
- 347 ページの『ELSE IF』
- 346 ページの『ELSE』
- **END IF** ステートメントの詳細については、 351 ページの『END (構造体)』

IF (論理)

目的

論理 **IF** ステートメントは論理式を計算し、その値が真であれば、指定したステートメントを実行します。

構文

```
►►─IF─(─logical_expr─)─stmt──────────────────────────────────────►◄
```

logical_expr
スカラー論理式です。

stmt ラベルが付なし実行可能ステートメントです。

規則

論理 **IF** ステートメントを実行すると、*logical_expr* が計算されます。 *logical_expr* の値が真であれば、*stmt* が実行されます。 *logical_expr* の値が偽であれば、*stmt* は実行されず、**IF** ステートメントは機能しません (**CONTINUE** ステートメントと同様になります)。

logical_expr 内の関数参照を実行すると、*stmt* 内の変数が変化する場合があります。

stmt を、 **SELECT CASE**、**CASE**、**END SELECT**、**DO**、 **DO WHILE**、**END DO**、ブロック **IF**、**ELSE IF**、**ELSE**、 **END IF**、**END FORALL**、他の論理 **IF**、**ELSEWHERE**、**END WHERE**、 **END**、**END FUNCTION**、 **END SUBROUTINE** ステートメント、**FORALL** 構造体ステートメント、または **WHERE** 構造体ステートメントにすることはできません。

例

```
IF (ERR.NE.0) CALL ERROR(ERR)
```

関連情報

147 ページの『第 6 章 制御構造』

IMPLICIT

目的

IMPLICIT ステートメントは、デフォルトの暗黙のタイプを変更または確認します。または **IMPLICIT NONE** の形式を指定すると、暗黙のタイプの設定規則をすべて無効にします。

構文

```
>>─IMPLICIT──┬─NONE──────────────────────────────────────┬─────────────────><
             │ ┌─,──────────────────────────────────────┐ │
             │ ▼                                         │ │
             └───┬─type_spec──────┬─(─range_list─)─┴─┘
                 │          (1)   │
                 ├─STATIC─────────┤
                 │             (2)│
                 ├─AUTOMATIC──────┤
                 │             (3)│
                 └─UNDEFINED──────┘
```

注:

1　　IBM 拡張

2　　IBM 拡張

3　　IBM 拡張

type_spec
: データ型を指定します。 481 ページの『タイプ宣言』を参照してください。

range
: 1 つの英字か、または英字の範囲です。英字の範囲は *letter*$_1$-*letter*$_2$ という形式で指定します。ここで、*letter*$_1$ は範囲の最初の英字、*letter*$_1$ に続く *letter*$_2$(アルファベット順) は範囲の最後の英字です。ドル記号 ($) と下線 (_) も範囲内で使用することができます。下線 (_) はドル記号 ($) の後に続き、ドル記号は Z の後に続きます。したがって、範囲 Y - _ は Y、Z、$、_ と同じことになります。

規則

文字の範囲をオーバーラップさせることはできません。つまり、特定の文字に対して複数のタイプを指定することはできません。

特定の有効範囲単位内で、**IMPLICIT** ステートメントの中で文字が指定されていない場合、プログラム単位またはインターフェース本体のエンティティーの暗黙タイプは、I-N で始まる文字の場合はデフォルトの整数、それ以外の文字の場合はデフォルトの実数で

す。内部プロシージャーまたはモジュール・プロシージャーのデフォルトは、ホスト有効範囲単位が使用する暗黙のタイプと同じです。

range_list によって指定した文字で始まるすべてのデータ・エンティティー名で、明示的にタイプを指定していないものに対しては、直前の *type_spec* で指定したタイプが与えられます。派生型がホストの有効範囲にアクセス可能な場合に、ローカルな有効範囲内でアクセス不能な派生型に対して暗黙のタイプが存在できることに注意してください。

IBM 拡張

STATIC または **AUTOMATIC** として指定する 1 つの文字または文字の範囲は、どのデータ型の **IMPLICIT** ステートメントにも指定することができます。 *range_list* 内の英字に対して、有効範囲単位内で *type_spec* と **UNDEFINED** の両方を指定することはできません。同じ文字に対して **STATIC** と **AUTOMATIC** の両方を指定することもできません。

IBM 拡張 の終り

ある有効範囲単位内で **IMPLICIT NONE** 形式を指定した場合、タイプ宣言ステートメントを使用してその有効範囲単位に対してローカルである名前のデータ型を指定しなければなりません。明示的に定義したデータ型を持たないシンボル名を参照することはできません。これにより、不注意で参照されるすべてのシンボル名を制御することができます。 **IMPLICIT NONE** を指定する場合、他の **IMPLICIT** ステートメントを同一の有効範囲単位内で指定することはできません。ただし、**STATIC** または **AUTOMATIC** を含んでいるステートメントは指定することができます。コンパイラー・オプション **-qundef** を指定してプログラムをコンパイルし、**IMPLICIT** ステートメントを使用できる有効範囲単位に指定した **IMPLICIT NONE** ステートメントと同じ効果を得ることができます。

IBM 拡張

IMPLICIT UNDEFINED は指定した文字または文字範囲に関する暗黙のデータ型のデフォルトをオフにします。 **IMPLICIT UNDEFINED** を指定した場合、指定した文字で始まるすべてのシンボル名のデータ型を範囲指定単位内で宣言する必要があります。有効範囲単位に対してローカルである各シンボル名のうち、データ型が明示的に定義されていないものには、コンパイラー診断メッセージが出されます。

IBM 拡張 の終り

IMPLICIT ステートメントによって、組み込み関数のデータ型が変わることはありません。

IBM 拡張

-qsave/ -qnosave コンパイラー・オプションを使用すると、ストレージ・クラスの事前定義規則を変更することができます。

-qsave コンパイラー・オプション	事前定義規則を作成します。	IMPLICIT STATIC(a - _)
-qnosave コンパイラー・オプション	事前定義規則を作成します。	IMPLICIT AUTOMATIC(a - _)

コンパイラー・オプション **-qmixed** を指定した場合でも、範囲のリスト項目の大文字・小文字は区別されません。たとえば、**-qmixed** を指定した場合、 `IMPLICIT INTEGER(A)` は `A` で始まるデータ・オブジェクトの暗黙のタイプ設定の他に、`a` で始まる暗黙のタイプ設定にも影響を与えます。

IBM 拡張 の終り

例

```
      IMPLICIT INTEGER (B), COMPLEX (D, K-M), REAL (R-Z,A)
!  This IMPLICIT statement establishes the following
!  implicit typing:
!
!        A: real
!        B: integer
!        C: real
!        D: complex
!   E to H: real
!     I, J: integer
!  K, L, M: complex
!        N: integer
!   O to Z: real
!        $: real
!        _: real
```

関連情報

- 暗黙の規則については、 69 ページの『タイプの決め方』
- 78 ページの『変数のストレージ・クラス』
- 「ユーザーズ・ガイド」の『**-qundef** オプション』
- 「ユーザーズ・ガイド」の『**-qsave** オプション』

INQUIRE

目的

INQUIRE ステートメントは名前付きファイルの特性、または特定の装置との接続状況に関する情報を取得します。

INQUIRE ステートメントには次の 3 つの形式があります。

- ファイルによる照会。この場合、**FILE=** 指定子が必要です。
- 出力リストによる照会。この場合、**IOLENGTH=** 指定子が必要です。
- 装置による照会。この場合、**UNIT=** 指定子が必要です。

構文

```
►►─INQUIRE──┬─(─inquiry_list─)─────────────────────────┬────────────────────────►◄
            └─(─IOLENGTH─=─iol─)─output_item_list─┘
```

iol
不定様式の出力ステートメント内の出力リストを使用した後の、データのバイト数を示します。 *iol* はスカラー整変数です。

output_item
PRINT または **WRITE** ステートメントを参照してください。

inquiry_list
INQUIRE ステートメントのファイルによる照会用および装置による照会用の照会指定子のリストです。ファイルによる照会の形式には装置指定子を指定することができず、装置による照会の形式にはファイル指定子を指定することができません。どの **INQUIRE** ステートメントにも 2 回以上指定子を指定することはできません。照会指定子には、次のものがあります。

[UNIT=] *u*
装置指定子です。この指定子は、装置による照会形式のステートメントで照会しようとしている装置を指定します。 *u* は、アスタリスク以外の値を持つ外部装置識別子でなければなりません。外部装置識別子はスカラー整数式 (0 ～ 2147483647 の値を持つ) で表される外部ファイルを参照します。オプションの文字である **UNIT=** を省略する場合は、*inquiry_list* の最初の項目は *u* でなければなりません。

IOSTAT= *ios*
I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。この指定子を含む I/O ステートメントの実行が完了すると、*ios* は以下の値に定義されます。
- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値

IOSTAT= 指定子をコーディングすると、エラー・メッセージは抑制されます。

ERR= *stmt_label*
エラーが発生した場合に制御が移される同じ有効範囲単位内の実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子をコーディングすると、エラー・メッセージは抑制されます。

FILE= *char_expr*

ファイル指定子です。これは、ファイルによる照会形式のステートメントで照会しようとしているファイルの名前を指定します。 *char_expr* はスカラー文字式で、この式の後続ブランクを除去した後の値は、有効な AIX オペレーティング・システム・ファイル名です。指定したファイルが存在している必要はなく、また装置と関連している必要もありません。

IBM 拡張

注: AIX オペレーティング・システム・ファイル名が有効であるためには、各ファイル名の長さが 255 文字以下で、絶対パス名の合計長が 1023 文字以下でなければなりません (ただし、絶対パス名は指定しなくてもかまいません)。

IBM 拡張 の終り

ACCESS= *char_var*
ファイル接続が直接アクセス用、順次アクセス用、ストリーム・アクセス用のいずれであるかを示します。 *char_var* はスカラー文字変数で、ファイルが順次アクセス用に接続されている場合は、値 **SEQUENTIAL** が割り当てられます。ファイルが直接アクセス用に接続されている場合は、値 **DIRECT** が割り当てられます。ファイルがストリーム・アクセス用に接続されている場合は、値 **STREAM** が割り当てられます。接続がない場合、*char_var* には値 **UNDEFINED** が割り当てられます。

FORM= *char_var*
ファイルが定様式 I/O 用に接続されているのか、不定様式 I/O 用に接続されているのかを示します。 *char_var* はデフォルトのスカラー文字変数で、ファイルが定様式 I/O 用に接続されている場合は、値 **FORMATTED** が割り当てられます。ファイルが不定様式 I/O 用に接続されている場合は、値 **UNFORMATTED** が割り当てられます。接続がない場合、*char_var* には値 **UNDEFINED** が割り当てられます。

IBM 拡張

ASYNCH= *char_variable*
装置が非同期アクセス用に接続されているかどうかを示します。

char_variable は、以下の値を戻す文字変数です。
- **YES**。装置が同期アクセスと非同期アクセスの両方用に接続されている場合。
- **NO**。装置が同期アクセス用にのみ接続されている場合。
- **UNDEFINED**。装置が接続されていない場合。

IBM 拡張 の終り

IBM 拡張

TRANSFER= *char_variable*
同期または非同期 (あるいはその両方) データ転送が、ファイルの転送方法として許可されるかどうかを示す非同期 I/O 指定子です。

char_variable は、スカラー文字変数です。 *char_variable* に値 **BOTH** が割り当てられる場合、同期および非同期データ転送の両方が許可されます。 *char_variable* に値 **SYNCH** が割り当てられる場合、同期データ転送のみが許可されます。 *char_variable* に値 **UNKNOWN** が割り当てられる場合、プロセッサーはこのファイルについて許可できる転送方法を判別できません。

IBM 拡張 の終り

POS=*integer_var*
integer_var はスカラー・デフォルト整変数で、ストリーム・アクセス用に接続されたファイルのファイル位置の値を示します。*integer_var* には、ストリーム・アクセス用に接続されたファイルの現行位置の直後にあるファイル記憶単位の番号が割り当てられます。ファイルが終端点にある場合、*integer_var* には、ファイル内で最も大きい番号のファイル記憶単位よりも 1 大きい値が割り当てられます。*integer_var* は、ファイルがストリーム・アクセス用に接続されている場合、または直前のエラー状態によりファイルの位置が決定できない場合は未定義になります。

RECL= *rcl*

直接アクセス用に接続されたファイルのレコード長の値、または、順次アクセス用に接続されたファイルの最大レコード長の値を示します。

IBM 拡張

rcl は、レコード長の値に割り当てられている、 **INTEGER(4)** タイプ、64 ビットの **INTEGER(8)** タイプ、またはデフォルトの整数タイプのスカラー変数です。

IBM 拡張 の終り

ファイルが定様式 I/O 用に接続されている場合、長さは文字データを含むすべてのレコードの文字数です。ファイルが不定様式 I/O 用に接続されている場合、長さはデータのバイト数で計られます。接続がない場合、*rcl* は未定義になります。

ファイルがストリーム・アクセス用に接続されている場合、*rcl* は未定義になります。

BLANK= *char_var*
定様式 I/O 用に接続されたファイルのブランクに関するデフォルト処理を示します。 *char_var* はスカラー文字変数で、数値入力フィールド内のブランクをすべて無視する場合、値 **NULL** が割り当てられます。先行ブランク以外をすべてゼロとして処理する場合、値 **ZERO** が割り当てられます。接続がない場合、あるいは接続が定様式 I/O 用ではない場合、*char_var* には値 **UNDEFINED** が割り当てられます。

EXIST= *ex*
ファイルまたは装置が存在しているかどうかを示します。 *ex* は **LOGICAL(4)** またはデフォルトの論理タイプのスカラー変数です。これには値 `true` または `false` が割り当てられます。ファイルによる照会形式のステートメントの場合、**FILE=** 指定子によって指定されたファイルが存在すれば値 `true` が割り当てられます。そのファイルが存在しなければ値 `false` が割り当てられます。装置による照会形式のステートメントの場合、**UNIT=** によって指定された装置が存在していると値 `true` が割り当てられます。装置が無効な場合は値 `false` が割り当てられます。

OPENED= *od*
ファイルまたは装置が接続されているかどうかを示します。 *od* は **LOGICAL(4)** またはデフォルトの論理タイプのスカラー変数です。これには値 `true` または `false` が割り当てられます。ファイルによる照会形式のステートメントの場合 **FILE=** *char_var* 指定子によって指定されたファイルが装置に接続されていれば、値 `true` が割り当てられます。ファイルが装置に接続されていなければ、値 `false` が割り当てられます。装置による照会形式のステートメントの場合、 **UNIT=** によって指定された装置がファイルに接続されていれば、値 `true` が割り当てられます。装置がファイルに接続されていなければ、値 `false` が割り当てられます。事前接続ファイルの中でクローズされていないものについては、最初の I/O 操作の前後でともに値 `true` が割り当てられます。

NUMBER= *num*
現在ファイルに関連付けられている外部装置識別子を示します。 *num* は **INTEGER(4)** またはデフォルト整数のタイプのスカラー変数です。これにはファイルに現在接続されている装置の外部装置識別子の値が割り当てられます。ファイルに接続されている装置がない場合、*num* には値 `-1` が割り当てられます。

NAMED= *nmd*
ファイルが名前を持っているかどうかを示します。 *nmd* は **LOGICAL(4)** またはデフォルトの論理タイプのスカラー変数です。ファイルが名前を持っていれば、値 `true` が割り当てられます。ファイルが名前を持っていなければ、値 `false` が割り当てられます。

NAME= *fn*
ファイルの名前を示します。 *fn* はスカラー文字変数で、装置が接続されているファイルの名前が割り当てられます。

SEQUENTIAL= *seq*
ファイルが順次アクセス用に接続されているかどうかを示します。 *seq* はスカラー文字変数で、ファイルに順次アクセスを行える場合は値 **YES**、ファイルに順次アクセスを行えない場合は値 **NO**、どちらとも決定できない場合は値 **UNKNOWN** が割り当てられます。

STREAM=*strm*
スカラー・デフォルト文字変数で、ファイルがストリーム・アクセス用に接続されているかを示します。 *strm* には、ファイルにストリーム・アクセスを行える場合は値 **YES**、ファイルにストリーム・アクセスを行えない場合は値 **NO**、どちらとも決定できない場合は値 **UNKNOWN** が割り当てられます。

DIRECT= *dir*
ファイルが直接アクセス用に接続されているかどうかを示します。 *dir* はスカラー文字変数で、ファイルに直接アクセスを行える場合は値 **YES**、ファイルに直接アクセスを行えない場合は値 **NO**、どちらとも決定できない場合は値 **UNKNOWN** が割り当てられます。

FORMATTED= *fmt*
ファイルが定様式 I/O 用に接続できるかどうかを示します。*fmt* はスカラー文字変数で、ファイルを定様式 I/O 用に接続できる場合には値 **YES**、ファイルを定様式 I/O 用に接続できない場合には値 **NO**、どちらとも決定できない場合には **UNKNOWN** が割り当てられます。

UNFORMATTED= *unf*
ファイルが不定様式 I/O 用に接続できるかどうかを示します。*fmt* はスカラー文字変数で、ファイルを不定様式 I/O 用に接続できる場合には、値 **YES** 、ファイルを不定様式 I/O 用に接続できない場合には値 **NO** 、どちらとも決定できない場合には **UNKNOWN** が割り当てられます。

NEXTREC= *nr*
直接アクセス用に接続されたファイル上のどこで次のレコードの読み取りまたは書き込みを実行できるかを示します。 *nr* は **INTEGER(4)**、**INTEGER(8)**、またはデフォルト整数のタイプのスカラー変数です。これには `n + 1` の値が割り当てられます。ここで、`n` は直接アクセス用に接続されたファイル上で最後に読み取られた、または書き込まれたレコードのレコード番号です。ファイルが接続されていても、接続後にレコードの読み取りまたは書き込みが実行され

ていない場合、*nr* には値 1 が割り当てられます。ファイルが直接アクセス用に接続されていない場合、あるいは、以前にエラーがあったために、ファイルの位置を決定できない場合、*nr* は未定義になります。

IBM 拡張

レコードの数は 2**31-1 よりも多くなることがあるため、**INTEGER(8)** の **NEXTREC=** 指定子でスカラー変数を指定させることも選択できます。 これを行うには、多くの方法がありますが、2 つの例を示します。
- *nr* を **INTEGER(8)** として明示的に宣言する。
- *-qintsize=8* コンパイラー・オプションでデフォルトの整数の kind を変更する。

IBM 拡張 の終り

POSITION= *pos*

ファイルの位置を示します。 *pos* はスカラー文字変数です。**OPEN** ステートメントによって初期点に位置決めされるようにファイルが接続されている場合は値 **REWIND**、ファイルの最後のレコードの前の位置、あるいは終端ポイントに位置決めされるようにファイルが接続されている場合には値 **APPEND**、位置を変えないようにファイルが接続されている場合には値 **ASIS**、接続がない場合、あるいはファイルが直接アクセス用に接続されていない場合には **UNDEFINED** がそれぞれ割り当てられます。

オープンされた後ファイルが初期点に再度位置決めされている場合、*pos* には値 **REWIND** が割り当てられます。オープンされた後ファイルの最後のレコードの直前に再度位置決めされている場合 (あるいは終端ポイントにファイルの最後のレコードがない場合)、*pos* には値 **APPEND** が割り当てられます。上記の 2 つが共に真で、ファイルが空のとき、 *pos* には **APPEND** が割り当てられます。ファイルがファイルの最後のレコードの後に位置決めされた場合、*pos* には **ASIS** が割り当てられます。

ACTION= *act*

ファイルが読み取りと書き込みあるいはそれらのどちらか一方のアクセスのうちいずれのために接続されているかを示します。*act* はスカラー文字変数です。ファイルが入力用だけに接続されている場合は値 **READ**、ファイルが出力用だけに接続されている場合は値 **WRITE**、ファイルが I/O 用に接続されている場合には値 **READWRITE**、接続がない場合には値 **UNDEFINED** がそれぞれ割り当てられます。

READ= *rd*

ファイルが読み取れるかどうかを示します。 *rd* はスカラー文字変数です。ファイルが読み取れる場合は **YES**、ファイルが読み取れない場合は **NO**、ファイルが読み取れるかどうか判断できない場合は値 **UNKNOWN** が割り当てられます。

WRITE= *wrt*
ファイルに書き込みができるかどうかを示します。 *wrt* はスカラー文字変数です。ファイルに書き込める場合は値 **YES**、ファイルが書き込めない場合は値 **NO**、ファイルに書き込めるかどうか判断できない場合は値 **UNKNOWN** が割り当てられます。

READWRITE= *rw*
ファイルに対して読み取りと書き込みの両方が実行できるかどうかを示します。 *rw* はスカラー文字変数です。ファイルに対して読み取りと書き込みの両方が実行できる場合は値 **YES**、ファイルが読み取りと書き込みの両方が実行できない場合は値 **NO**、ファイルに対して読み取りと書き込みの両方が実行できるかどうか判断できない場合は値 **UNKNOWN** が割り当てられます。

DELIM= *del*
リスト指示形式設定または名前リスト形式設定により書き込まれた文字データを区切る書式を使用する場合に、その書式を指定します。 *del* はスカラー文字変数です。データの区切りにアポストロフィを使用する場合は値 **APOSTROPHE**、データの区切りに引用符を使用する場合は値 **QUOTE**、データの区切りにアポストロフィも引用符も使用しない場合は値 **NONE**、そして、ファイルの接続または定様式データへの接続がない場合は値 **UNDEFINED** が割り当てられます。

PAD= *pd*
ファイルの接続が **PAD=NO** を指定しているかどうかを示します。 *pd* はスカラー文字変数です。ファイルの接続が **PAD=NO** を指定している場合には値 **NO**、その他の場合には値 **YES** が割り当てられます。

SIZE=*filesize*
filesize はスカラー整変数で、ファイル・サイズ (バイト単位) が割り当てられます。

規則

INQUIRE ステートメントを実行できるのは、ファイルが装置と関連する前、関連している間、または関連した後です。 **INQUIRE** ステートメントの結果として割り当てられる値は、すべてステートメントが実行される時点での現在値です。

IBM 拡張

装置またはファイルが接続されている場合、**ACCESS=**、**SEQUENTIAL=**、**STREAM=**、**DIRECT=**、**ACTION=**、**READ=**、**WRITE=**、**READWRITE=**、**FORM=**、**FORMATTED=**、**UNFORMATTED=**、**BLANK=**、**DELIM=**、**PAD=**、**RECL=**、**POSITION=**、**NEXTREC=**、**NUMBER=**、**NAME=**、および **NAMED=** 指定子に戻される値は接続の特性であり、ファイルの特性ではありません。 **EXIST=** と **OPENED=** 指定子は、この状態では `true` を戻すことに注意してください。

装置またはファイルが接続されていない場合、あるいは存在しない場合、 **ACCESS=**、**ACTION=**、**FORM=**、**BLANK=**、**DELIM=**、**POSITION=** 指定子は値 **UNDEFINED** を戻し、**DIRECT=**、**SEQUENTIAL=**、**STREAM=**、**FORMATTED=**、**UNFORMATTED=**、**READ=**、**WRITE=** および **READWRITE=** 指定子は値 **UNKNOWN** を戻し、**RECL=** および **NEXTREC=** 指定子変数は定義されず、**PAD=** 指定子は値 **YES** を戻し、**OPENED** 指定子は値 `false` を戻します。 **SIZE=** 指定子によって戻される値は -1 です。

装置またはファイルが存在しない場合、 **EXIST=** と **NAMED=** 指定子は値 `false` を戻し、**NUMBER=** 指定子は値 -1 を戻し、 **NAME=** 指定子変数は定義されません。

装置またはファイルが存在していても接続していない場合、**EXIST=** 指定子は値 `true` を戻します。装置による照会形式のステートメントの場合、 **NAMED=** 指定子は値 `false` を戻し、 **NUMBER=** 指定子は装置番号を戻し、**NAME=** 指定子変数は定義されません。ファイルによる照会形式のステートメントの場合、**NAMED=** 指定子は値 `true` を戻し、**NUMBER=** 指定子は値 -1 を戻し、**NAME=** 指定子はファイル名を戻します。

IBM 拡張 の終り

同一の **INQUIRE** ステートメント内の複数の指定子に同じ変数名を指定することはできません。また、同じ変数名を指定子リスト上の他の変数と関連させることもできません。

例

```
SUBROUTINE SUB(N)
  CHARACTER(N) A(5)
  INQUIRE (IOLENGTH=IOL) A(1)  ! Inquire by output list
  OPEN (7,RECL=IOL)
   ⋮
END SUBROUTINE
```

関連情報

- 229 ページの『条件および IOSTAT 値』
- 217 ページの『第 8 章 I/O の概念』

INTEGER

目的

INTEGER タイプ宣言ステートメントは、整数タイプのオブジェクトと関数の長さと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>─INTEGER─┬───────────────┬─┬─::──────────────────────┬─entity_decl_list─────────><
           └─kind_selector─┘ └─,─attr_spec_list─::─┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

kind_selector

```
>>─┬─(─┬──────────┬─int_initialization_expr─)─┬──────────────────────────><
   │   └─KIND─ = ─┘                            │
   │                        (1)                │
   └─ * ─int_literal_constant──────────────────┘
```

注:

1 IBM 拡張

IBM 拡張

整数エンティティーの長さ (1, 2, 4, 8) を指定します。 *int_literal_constant* には、kind 型付きパラメーターは指定できません。

IBM 拡張 の終り

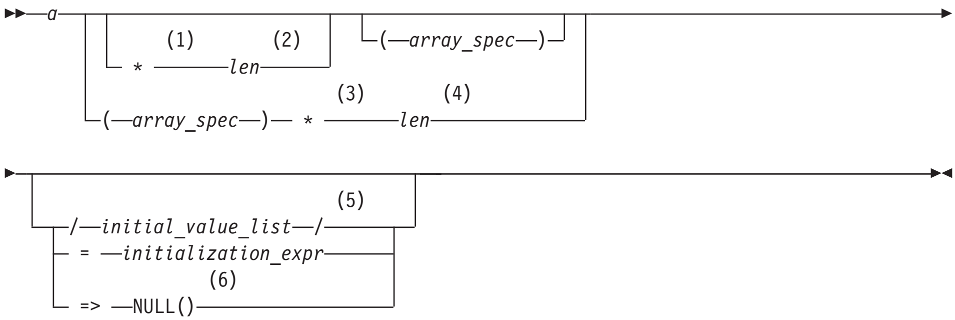
attr_spec
特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
IN、**OUT**、または **INOUT** のいずれかです。

:: ダブル・コロン・セパレーターです。複数の属性を指定する場合、あるいは *=initialization_expr* または F95 => **NULL()** F95 を使用する場合に必要になります。

array_spec
次元境界のリストです。

entity_decl



注:

1 IBM 拡張

2 IBM 拡張

3 IBM 拡張

4 IBM 拡張

5 IBM 拡張

6 Fortran 95

a オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

len *kind_selector* に指定されている長さをオーバーライドします。kind 型付きパラメーターを指定することはできません。エンティティーの長

さは、許容できる長さ指定の 1 つを表す整数のリテラル定数でなければなりません。

IBM 拡張 の終り

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。オブジェクトがブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、 IBM または モジュール内の名前付き共通ブロックにある場合、そのオブジェクトを初期化することができます。 IBM

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、*initialization_expr* F95 または **NULL()** F95 を指定した場合、変数は最初に定義されます。

Fortran 95

宣言するエンティティーが派生型コンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルト初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、 *a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている `T` または `F` がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

例

```
MODULE INT
  INTEGER, DIMENSION(3) :: A,B,C
  INTEGER :: X=234,Y=678
END MODULE INT
```

関連情報

- 29 ページの『整数』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

INTENT

目的

INTENT 属性は仮引き数の意図的な使用を指定します。

構文

```
►►─INTENT─(─┬─IN────┬─)─┬────┬─dummy_arg_name_list─────────────────────►◄
            ├─OUT───┤   └─::─┘
            └─INOUT─┘
```

dummy_arg_name
仮引き数の名前です。これに仮プロシージャーを指定することはできません。

規則

非ポインターの割り振り不可の仮引き数を指定した場合、**INTENT** 属性は以下の特性を持ちます。

- **INTENT(IN)** は、サブプログラムの実行中に仮引き数が再定義されないこと、あるいは未定義にならないことを指定します。
- **INTENT(OUT)** は、サブプログラムの中で参照される前に仮引き数を定義しなければならないことを指定します。このような仮引き数はサブプログラムの呼び出し時に未定義にならない場合があります。
- **INTENT(INOUT)** は、仮引き数と呼び出し中のサブプログラムとの間でデータをやりとりできることを指定します。

ポインター仮引き数を指定した場合、**INTENT** 属性は以下の特性を持ちます。

- **INTENT(IN)** は、プロシージャーの実行中は、ポインターのターゲットの割り振りが解除されない限り、ポインター仮引き数の関連付け状況を変更できないことを指定します。ポインターのターゲットの割り振りが解除された場合、ポインター仮引き数の関連付け状況は未定義になります。

  INTENT(IN) ポインター仮引き数を、ポインター割り当てステートメント内でポインター・オブジェクトとして使用することはできません。 **INTENT(IN)** ポインター仮引き数の割り振り、割り振り解除、またはヌル文字化を行うことはできません。

  関連する仮引き数が **INTENT(OUT)** または **INTENT(INOUT)** 属性を持つポインターである場合、**INTENT(IN)** ポインター仮引き数をプロシージャーの実引き数として指定することはできません。
- **INTENT(OUT)** は、プロシージャーを実行した時点でポインター仮引き数の関連付け状況が未定義であることを指定します。
- **INTENT(INOUT)** は、仮引き数と呼び出し中のサブプログラムとの間でデータをやりとりできることを指定します。

割り振り可能仮引き数を指定した場合、**INTENT** 属性は以下の特性を持ちます。

- **INTENT(IN)** は、プロシージャーの実行中は仮引き数の割り振り状況を変更できないこと、およびこれを再定義するか未定義にしてはならないことを指定します。
- **INTENT(OUT)** は、関連する実引き数が割り振られている場合は、プロシージャーを実行した時点でこの実引き数が割り振り解除されることを指定します。
- **INTENT(INOUT)** は、仮引き数と呼び出し中のサブプログラムとの間でデータをやりとりできることを指定します。

ポインターまたは割り振り可能仮引き数に **INTENT** 属性を指定しない場合、この仮引き数の使用は、関連する実引き数による制限および制約を受けます。

意図的な **OUT** または **INOUT** を指定して仮引き数と関連付けられる実引き数は定義可能でなければなりません。したがって、意図的な **IN** を指定した仮引き数、または定数である実引き数、定数のサブオブジェクト、あるいは式を、意図的な **OUT** または **INOUT** を指定する引き数を必要とするサブプログラムに実引き数として渡すことはできません。

ベクトル添え字を持つ配列セクションである実引き数を、定義または再定義された (すなわち **OUT** または **INOUT** という意図を持つ) 仮配列と関連させることはできません。

INTENT 属性と互換性のある属性

- ALLOCATABLE
- DIMENSION
- OPTIONAL
- POINTER
- TARGET
- VALUE
- VOLATILE

VALUE 属性は、意図的な **IN** を指定した仮引き数にのみ使用できます。

IBM 拡張

言語間呼び出しに使用される **%VAL** 組み込み関数は、意図的な **IN** を指定した仮引き数、または意図が指定されていない仮引き数に対応する実引き数にのみ使用することができます。この制約は **%REF** 組み込み関数には適用されません。

IBM 拡張 の終り

例

```
PROGRAM MAIN
  DATA R,S /12.34,56.78/
  CALL SUB(R+S,R,S)
END PROGRAM

SUBROUTINE SUB (A,B,C)
  INTENT(IN) A
  INTENT(OUT) B
  INTENT(INOUT) C
  C=C+A+ABS(A)            ! Valid references to A and C
                          ! Valid redefinition of C
  B=C**2                  ! Valid redefinition of B
END SUBROUTINE
```

関連情報

- 199 ページの『仮引き数の意図』
- 196 ページの『引き数関連付け』
- 言語間呼び出しの詳細については、 197 ページの『%VAL および %REF』
- 195 ページの『仮引き数』

INTERFACE

目的

INTERFACE ステートメントは、インターフェース・ブロックの最初のステートメントです。これは外部または仮プロシージャーに対して明示インターフェースを指定できます。

構文

```
>>──INTERFACE──┬──────────────┬──────────────────────────────────><
               └─generic_spec─┘
```

generic_spec

```
>>──┬─generic_name─────────────────────┬─────────────────────────><
    ├─OPERATOR──(──defined_operator──)─┤
    └─ASSIGNMENT──(── = ──)────────────┘
```

defined_operator
定義済み単項演算子、定義済み 2 進演算子、または拡張組み込み演算子です。

規則

generic_spec を指定した場合、インターフェース・ブロックは総称になります。
generic_spec を指定しない場合、インターフェース・ブロックは非総称になります。
generic_name は、インターフェース・ブロック内のすべてのプロシージャーを参照する単一名を指定します。総称名でのプロシージャー参照が発生するたびに、最大 1 つの特定のプロシージャーが呼び出されます。

Fortran 95

INTERFACE ステートメントに *generic_spec* を指定する場合、それは対応する **END INTERFACE** ステートメント内の *generic_spec* と一致しなければなりません。

INTERFACE ステートメント内の *generic_spec* が *generic_name* である場合、対応する **END INTERFACE** ステートメントの *generic_spec* は、同じ *generic_name* でなければなりません。

Fortran 95 の終り

INTERFACE ステートメントに *generic_spec* を指定しない場合、*generic_spec* のあるなしに関係なく、すべての **END INTERFACE** ステートメントと一致させることができます。

プロシージャーの 1 つの有効範囲単位内に明示インターフェースを複数指定することはできません。

アクセス可能であれば、特定のインターフェースを介していつでもプロシージャーを参照することができます。プロシージャーに総称インターフェースが存在する場合は、その総称インターフェースを介してプロシージャーを参照することができます。

generic_spec が **OPERATOR**(*defined_operator*) の場合、インターフェース・ブロックは定義済みの演算子を定義したり、組み込み演算子を拡張したりできます。

generic_spec が **ASSIGNMENT**(=) の場合、インターフェース・ブロックは組み込み割り当てを拡張できます。

例

```
INTERFACE                            ! Nongeneric interface block
  FUNCTION VOL(RDS,HGT)
    REAL VOL, RDS, HGT
  END FUNCTION VOL
  FUNCTION AREA (RDS)
    REAL AREA, RDS
  END FUNCTION AREA
END INTERFACE

INTERFACE OPERATOR (.DETERMINANT.)   ! Defined operator interface
```

```
  FUNCTION DETERMINANT(X)
    INTENT(IN) X
    REAL X(50,50), DETERMINANT
  END FUNCTION
END INTERFACE

INTERFACE ASSIGNMENT(=)              ! Defined assignment interface
  SUBROUTINE BIT_TO_NUMERIC (N,B)
    INTEGER, INTENT(OUT) :: N
    LOGICAL, INTENT(IN)  :: B(:)
  END SUBROUTINE
END INTERFACE
```

関連情報

- 172 ページの『明示的インターフェース』
- 122 ページの『拡張組み込みおよび定義済み演算』
- 179 ページの『定義済み演算子』
- 180 ページの『定義済み割り当て』
- 378 ページの『FUNCTION』
- 473 ページの『SUBROUTINE』
- 417 ページの『MODULE PROCEDURE』
- 190 ページの『プロシージャー参照』
- 同じ総称名を持つ 2 つのプロシージャーを区別する方法の規則については、 176 ページの『明白な総称プロシージャー参照』

INTRINSIC

目的

INTRINSIC 属性は、名前を組み込みプロシージャーとして識別し、組み込みプロシージャーの特定名を実引き数として使用できるようにします。

構文

```
►►──INTRINSIC──┬──────┬──name_list─────────────────────────────────────►◄
               └─ ::─┘
```

name　組み込みプロシージャーの名前です。

規則

ある有効範囲単位内で組み込みプロシージャーの特定名を実引き数として使用する場合、その特定名は **INTRINSIC** 属性を持たなければなりません。総称名に **INTRINSIC** 属性を持たせることはできますが、総称名が特定名でないかぎり、引き数として渡すことはできません。

INTRINSIC 属性を持つ総称プロシージャーまたは特定プロシージャーは、それぞれの特性を保持します。

INTRINSIC 属性を持つ総称組み込みプロシージャーが、同時に総称インターフェース・ブロックの名前でもある場合があります。総称インターフェース・ブロックは総称組み込みプロシージャーに対する拡張機能を定義します。

INTRINSIC 属性と互換性のある属性

- PRIVATE
- PUBLIC

例

```
PROGRAM MAIN
  INTRINSIC SIN, ABS
  INTERFACE ABS
    LOGICAL FUNCTION MYABS(ARG)
      LOGICAL ARG
    END FUNCTION
  END INTERFACE

  LOGICAL LANS,LVAR
  REAL(8) DANS,DVAR
  DANS = ABS(DVAR)            ! Calls the DABS intrinsic procedure
  LANS = ABS(LVAR)            ! Calls the MYABS external procedure

! Pass intrinsic procedure name to subroutine
  CALL DOIT(0.5,SIN,X)        ! Passes the SIN specific intrinsic
END PROGRAM

SUBROUTINE DOIT(RIN,OPER,RESULT)
  INTRINSIC :: MATMUL
  INTRINSIC    COS
  RESULT = OPER(RIN)
END SUBROUTINE
```

関連情報

- 総称組み込みプロシージャーおよび特定組み込みプロシージャーは、 539 ページの『第 12 章 組み込みプロシージャー』にリストされています。特定の組み込み名が実引き数として使用されている場合は、この項を参照してください。
- 176 ページの『総称インターフェース・ブロック』

LOGICAL

目的

LOGICAL タイプ宣言ステートメントは、論理タイプのオブジェクトと関数の、長さと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>─LOGICAL──┬───────────────┬──┬─::─────────────────────┬─entity_decl_list─────────><
            └─kind_selector─┘  └─,─attr_spec_list─::─┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

kind_selector

```
>>──┬─(─┬──────────┬─int_initialization_expr─)─┬─────────────────────><
    │   └─KIND── = ┘                           │
    │                              (1)         │
    └─ * ─int_literal_constant─────────────────┘
```

注:

1　　IBM 拡張

IBM 拡張

論理エンティティーの長さ (1、2、4、8) を指定します。*int_literal_constant* には、kind 型付きパラメーターは指定できません。

IBM 拡張 の終り

attr_spec
: 特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
: **IN**、**OUT**、または **INOUT** のいずれかです。

::
: ダブル・コロン・セパレーターです。複数の属性を指定する場合、あるいは= *initialization_expr* F95 または **NULL()** F95 を使用する場合に必要になります。

array_spec
: 次元境界のリストです。

entity_decl

```
>>─a─┬─┬──────────────────┬─┬──────────────────┬─┬──────────────────────>
     │ │     (1)     (2)  │ └─(──array_spec──)─┘ │
     │ └─ * ──────────len─┘                      │
     │                        (3)      (4)       │
     └─(──array_spec──)── * ────────len──────────┘

>─┬──────────────────────────────────┬───────────────────────────────────><
  │                           (5)    │
  └─┬─/──initial_value_list──/───────┤
    ├─ = ──initialization_expr───────┤
    │                  (6)           │
    └─ => ──NULL()───────────────────┘
```

注:

1 IBM 拡張

2 IBM 拡張

3 IBM 拡張

4 IBM 拡張

5 IBM 拡張

6 Fortran 95

a
: オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

len *kind_selector* に指定されている長さをオーバーライドします。kind 型付きパラメーターを指定することはできません。エンティティーの長さは、許容できる長さ指定の 1 つを表す整数のリテラル定数でなければなりません。

IBM 拡張 の終り

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。ブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、またはモジュール内の名前付き共通ブロックにある場合、オブジェクトは初期化することができます。

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、*initialization_expr* F95 または **NULL()** F95 を指定した場合、変数は最初に定義されます。

Fortran 95

宣言するエンティティーが派生型コンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルト初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、 *a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている `T` または `F` がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

例

```
LOGICAL, ALLOCATABLE :: L(:,:)
LOGICAL :: Z=.TRUE.
```

関連情報

- 36 ページの『論理』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』

- 初期値の詳細については、 327 ページの『DATA』

MODULE

目的

MODULE ステートメントはモジュール・プログラム単位の最初のステートメントです。これには、他のプログラム単位へのアクセスを可能にする仕様と定義が含まれます。

構文

```
►►──MODULE──module_name──────────────────────────────────────────────────►◄
```

規則

モジュール名は、モジュール共通エンティティーにアクセスするために他のプログラム単位の中で **USE** ステートメントを使用して参照される、グローバル・エンティティーです。モジュール名として、他のプログラム単位の名前、外部プロシージャーの名前、あるいはプログラム内の共通ブロックの名前を使用することはできません。モジュール内のローカル名を使用することもできません。

モジュールを完了させる **END** ステートメントでモジュール名を指定する場合、そのモジュール名は **MODULE** ステートメントで指定したものと同じでなければなりません。

例

```
MODULE MM
   CONTAINS
     REAL FUNCTION SUM(CARG)
       COMPLEX CARG
       SUM_FNC(CARG) = IMAG(CARG) + REAL(CARG)
       SUM = SUM_FNC(CARG)
       RETURN
     ENTRY AVERAGE(CARG)
       AVERAGE = SUM_FNC(CARG) / 2.0
     END FUNCTION SUM
     SUBROUTINE SHOW_SUM(SARG)
       COMPLEX SARG
       REAL SUM_TMP
  10   FORMAT('SUM:',E10.3,' REAL:',E10.3,' IMAG',E10.3)
       SUM_TMP = SUM(CARG=SARG)
       WRITE(10,10) SUM_TMP, SARG
     END SUBROUTINE SHOW_SUM
END MODULE MM
```

関連情報

- 183 ページの『モジュール』
- 488 ページの『USE』
- 166 ページの『使用関連付け』
- **END MODULE** ステートメントの詳細については、 350 ページの『END』
- 438 ページの『PRIVATE』
- 442 ページの『PROTECTED』
- 444 ページの『PUBLIC』

MODULE PROCEDURE

目的

MODULE PROCEDURE ステートメントは総称インターフェースを持つモジュール・プロシージャーをリストします。

構文

```
▶▶──MODULE PROCEDURE──procedure_name_list──────────────────────────▶◀
```

規則

Fortran 95

MODULE PROCEDURE ステートメントは、総称仕様を持つインターフェース・ブロック内のインターフェース本体のどこにでも置くことができます。

Fortran 95 の終り

MODULE PROCEDURE は、モジュール・プロシージャーとして *procedure_name* にアクセス可能な有効範囲単位に含まれていなければなりません。また、この有効範囲単位でアクセス可能な名前でなければなりません。

procedure_name は、事前にインターフェース・ブロックで名前を指定するか、あるいは使用関連付けやホスト関連付けを使用して、それが指定されるインターフェース・ブロックの総称仕様と事前に関連付けられていてはなりません。

モジュール・プロシージャーの特性は、インターフェース本体ではなくモジュール・プロシージャー定義により決定されます。

例

```
MODULE M
  CONTAINS
  SUBROUTINE S1(IARG)
    IARG=1
  END SUBROUTINE
  SUBROUTINE S2(RARG)
    RARG=1.1
  END SUBROUTINE
END MODULE

USE M
INTERFACE SS
  SUBROUTINE SS1(IARG,JARG)
  END SUBROUTINE
  MODULE PROCEDURE S1, S2
END INTERFACE
CALL SS(N)                    ! Calls subroutine S1 from M
CALL SS(I,J)                  ! Calls subroutine SS1
END
```

関連情報

- 172 ページの『インターフェース・ブロック』
- 407 ページの『INTERFACE』
- 183 ページの『モジュール』

NAMELIST

目的

NAMELIST ステートメントは、 **READ**、**WRITE**、および **PRINT** ステートメントで使用する名前のリストを 1 つ以上指定します。

構文

Nname 名前リストのグループ名です。

variable_name
: 非定数境界を持つ配列仮引き数、非定数文字長を持つ変数、自動オブジェクト、ポインター、最終コンポーネントがポインターであるタイプの変数、割り振り可能オブジェクト、またはポインティング先であってはなりません。

規則

名前リスト・グループ名に属する名前のリストは、別の名前リスト・グループ名が現れるか、または **NAMELIST** ステートメントの終わりに達した時点で終了します。

variable_name は、使用関連付けまたはホスト関連付けを介してアクセスするか、同じ有効範囲単位内で前にある仕様ステートメントまたは暗黙の入力規則で、タイプおよび型付きパラメーターを指定している必要があります。暗黙にタイプが指定された場合、続くタイプ宣言ステートメントの中にあるオブジェクトの指定では、暗黙のタイプおよび型付きパラメーターを確認しなければなりません。名前リスト・グループ名を指定している名前リストの I/O ステートメントが存在する有効範囲単位内において、最終的にオブジェクトに含まれるコンポーネントへのアクセスができない場合、派生型のオブジェクトをリスト項目として指定することはできません。

variable_name は 1 つ以上の名前リストに属していてもかまいません。名前リスト・グループ名が **PUBLIC** 属性を持つ場合は、リスト内のどの項目も **PRIVATE** 属性または PRIVATE コンポーネントを持つことはできません。

Nname は、有効範囲単位内の複数の **NAMELIST** ステートメントに指定することができます。また、各 **NAMELIST** ステートメントに複数回指定することもできます。ある有効範囲単位内の同一の *Nname* の後に続く *variable_name_list* は、その *Nname* 用のリストの続きとして処理されます。

名前リストの名前は I/O ステートメントにだけ指定することができます。名前リスト・データの I/O 変換の規則はデータ変換の規則と同じです。

例

```
DIMENSION X(5), Y(10)
NAMELIST /NAME1/ I,J,K
NAMELIST /NAME2/ A,B,C /NAME3/ X,Y
WRITE (10, NAME1)
PRINT NAME2
```

関連情報

- 274 ページの『名前リストの形式設定』
- 「ユーザーズ・ガイド」の『*実行時オプションの設定*』

NULLIFY

目的

NULLIFY ステートメントは、ポインターを関連解除します。

構文

```
▶▶─NULLIFY─(─pointer_object_list─)──────────────────────────────────▶◀
```

pointer_object
: ポインター変数名または構造体コンポーネントです。

規則

pointer_object は **POINTER** 属性を持っていなければなりません。

> **ヒント**
> ポインターの初期化は常に、**NULLIFY** ステートメント、ポインター割り当て、 F95 デフォルト初期化 F95 、または明示的な初期化のいずれかで行います。

例

```
TYPE T
  INTEGER CELL
  TYPE(T), POINTER :: NEXT
ENDTYPE T
TYPE(T) HEAD, TAIL
TARGET :: TAIL
HEAD%NEXT => TAIL
NULLIFY (TAIL%NEXT)
END
```

関連情報

- 142 ページの『ポインターの割り当て』
- 167 ページの『ポインター関連付け』

OPEN

目的

OPEN ステートメントは既存の外部ファイルを装置に接続するため、事前結合された外部ファイルを作成し、それを装置に接続するため、あるいは外部ファイルと装置の間の接続に関する特定の指定子を変更するために使用できます。

構文

```
▶▶──OPEN──(──open_list──)──────────────────────────────────▶◀
```

open_list

装置指定子 (**UNIT=***u*) を必ず 1 つ含んでいなければならないリストです。このリストには、許可されている他の指定子をそれぞれ 1 つずつ入れることができます。有効な指定子は次のとおりです。

[UNIT=] *u*

装置指定子です。*u* は外部装置識別子で、その値はアスタリスクであってはなりません。外部装置識別子はスカラー整数式 (0 ～ 2,147,483,647 の値を持つ) で表される外部ファイルを示します。オプションの文字である **UNIT=** を省略する場合は、*u* は *open_list* の最初の項目でなければなりません。

IBM 拡張

ASYNCH= *char_expr*

明示的に接続された装置が非同期 I/O 用に使用されるかどうかを示す、非同期 I/O 指定子です。

char_expr はスカラー文字式で、その値は **YES** または **NO** のいずれかになります。 **YES** は、この接続に非同期データ転送ステートメントを許可することを指定します。 **NO** は、この接続に非同期データ転送ステートメントを許可しないことを指定します。指定される値は、そのファイルに対して許可される一連の転送方法になります。この指定子が省略される場合、デフォルト値は **NO** です。

事前接続された装置は、**ASYNCH=** に値 **NO** を指定して接続されています。

暗黙接続される装置の **ASYNCH=** 値は、装置上で実行される最初のデータ転送ステートメントによって決まります。最初のステートメントが非同期データ転送を実行し、暗黙接続されるファイルが非同期データ転送を許可する場合、**ASYNCH=** 値は **YES** になります。そうでない場合、**ASYNCH=** 値は **NO** になります。

IBM 拡張 の終り

IOSTAT= *ios*

I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。この指定子を含む I/O ステートメントの実行が完了すると、*ios* は以下の値に定義されます。

- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値

ERR= *stmt_label*

エラーが発生した場合に制御が移される、同じ有効範囲単位内の実行可能ステ

ートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子は、エラー・メッセージを抑制します。

FILE= *char_expr*

指定した装置に接続するファイルの名前を指定するファイル指定子です。

IBM 拡張

char_expr はスカラー文字式で、この式の後続ブランクを除去した後の値は、有効な AIX オペレーティング・システム・ファイル名です。 ファイル指定子が必要であるときにそれを省略した場合は、装置は (デフォルトにより) **fort.***u* に暗黙に接続されます。ここで、*u* は先行ゼロを除去して指定した装置です。暗黙接続されるファイルに代替ファイル名を使用できるようにするには、**UNIT_VARS** 実行時オプションを使用してください。

注: AIX オペレーティング・システム・ファイル名が有効であるためには、各ファイル名の長さが 255 文字以下で、絶対パス名の合計長が 1023 文字以下でなければなりません (ただし、絶対パス名は指定しなくてもかまいません)。

IBM 拡張 の終り

STATUS= *char_expr*

ファイルのオープン後の状況を指定します。 *char_expr* はスカラー文字式で、この式の後続ブランクを除去した後の値は、以下のうちの 1 つです。

- **OLD**。既存ファイルを装置に接続します。 **OLD** を指定する場合、ファイルが存在していなければなりません。ファイルが存在していない場合は、エラーが発生します。
- **NEW**。新しいファイルを作成して、装置に接続し、状況を **OLD** に変更します。 **NEW** を指定する場合、ファイルが存在していてはいけません。ファイルが存在している場合は、エラーが発生します。
- **SCRATCH**。切り離し時に削除される新しいファイルを作成して接続します。 **SCRATCH** を名前付きファイルと一緒に指定することはできません (つまり、**FILE=***char_expr* は省略しなければなりません)。
- **REPLACE**。ファイルが存在しない場合、ファイルが作成され、状態は **OLD** に変化します。ファイルが存在する場合、ファイルが削除され、同じ名前で新しいファイルが作成され、状態は **OLD** に変化します。
- **UNKNOWN**。既存のファイルを接続するか、または新しいファイルを作成し、接続します。ファイルが存在している場合は、**OLD** として接続されます。ファイルが存在していない場合は、**NEW** として接続されます。

デフォルトは **UNKNOWN** です。

ACCESS= *char_expr*

ファイルの接続のためのアクセス方式を指定します。 *char_expr* はスカラー文

字式で、式の値は、後続ブランクを除去すると、**SEQUENTIAL**、**DIRECT**、または **STREAM** のいずれかになります。デフォルトは **SEQUENTIAL** です。**ACCESS=** が **DIRECT** の場合は、**RECL=** を指定しなければなりません。**ACCESS=** が **STREAM** の場合は、**RECL=** を指定しなければなりません。

FORM= *char_expr*
ファイルを定様式 I/O 用に接続するのか、あるいは不定様式 I/O 用に接続するのかを指定します。 *char_expr* はスカラー文字式で、式の値は、後続ブランクを除去すると **FORMATTED** または **UNFORMATTED** のいずれかになります。ファイルを順次アクセス用に接続する場合、デフォルトは **FORMATTED** です。ファイルを直接アクセス用またはストリーム・アクセス用に接続する場合、デフォルトは **UNFORMATTED** です。

RECL= *integer_expr*

直接アクセス用に接続するファイル内の各レコードの長さ、あるいは順次アクセス用に接続するファイル内のレコードの最大長を指定します。 *integer_expr* はスカラー整数式で、式の値は正でなければなりません。ファイルを直接アクセス用に接続する場合、この指定子を指定しなければなりません。定様式 I/O の場合、長さは文字データを含むすべてのレコードの文字数です。不定様式 I/O の場合、長さはデータの内部形式に必要なバイト数です。不定様式の順次レコードの長さには、そのデータを取り囲む 4 バイト・フィールドは計算に入れません。

IBM 拡張

ファイルを 32 ビットの順次アクセス用に接続するときに **RECL=** を省略した場合、その長さは 2**31 からレコード・ターミネーターの長さを引いたものです。 32 ビットの定様式順次ファイルの場合、デフォルトのレコードの長さは 2**31-1 です。 32 ビットでアクセス可能な不定様式ファイルの場合、デフォルトのレコードの長さは 2**31-8 です。 32 ビットで無作為にアクセスできないファイルの場合、デフォルトの長さは **32,768** です。

ファイルを 64 ビットの順次アクセス用に接続するときに **RECL=** を省略した場合、その長さは 2**64 からレコード・ターミネーターの長さを引いたものです。 **UWIDTH** 実行時オプションが 64 に設定されるとき、64 ビットの定様式順次ファイルの場合、デフォルトのレコードの長さは、2**64-1 です。不定様式のファイルの場合、デフォルトのレコードの長さは 2**64-16 です。

IBM 拡張 の終り

BLANK= *char_expr*
形式仕様を使用する場合の、ブランクのデフォルトの解釈を制御します。 *char_expr* はスカラー文字式で、式の値は、後続ブランクを除去すると **NULL** または **ZERO** のいずれかになります。 **BLANK=** を指定する場合、

FORM='FORMATTED' を指定しなければなりません。 **BLANK=** を指定しないで、**FORM='FORMATTED'** を指定した場合、デフォルトは **NULL** です。

POSITION= *char_expr*
順次アクセス用またはストリーム・アクセス用に接続されたファイルのファイル位置を指定します。以前に存在していなかったファイルは、その初期点に位置付けられます。 *char_expr* はスカラー文字式で、式の値は、後続ブランクを除去すると、**ASIS**、**REWIND**、**APPEND** のいずれかになります。 **REWIND** はファイルを初期点に位置付けます。**APPEND** はファイルをファイル終了レコードの前に位置付けます。ファイル終了レコードが存在しない場合は、終端点に位置付けます。 **ASIS** は位置を変更しません。以下の場合を除き、デフォルト値は **ASIS** です。

- **OPEN** ステートメントの後の装置を参照する I/O ステートメント (**INQUIRE** ステートメントを除く) が **WRITE** ステートメントであり、かつ
  - **STATUS=** 指定子が **UNKNOWN** で、**-qposition** コンパイラー・オプションが **appendunknown** を指定しているか。
  - **STATUS=** 指定子が **OLD** で、**-qposition** コンパイラー・オプションが **appendold** を指定している。

このような場合、**WRITE** ステートメントが実行されると、**POSITION=** 指定子のデフォルト値は **APPEND** になります。

ACTION= *char_expr*
認められる I/O 操作を指定します。 *char_expr* はスカラー文字式で、その値は、**READ**、**WRITE**、または **READWRITE** となります。 **READ** を指定した場合、**WRITE** および **ENDFILE** ステートメントはこの接続を参照できません。 **WRITE** を指定した場合、**READ** ステートメントはこの接続を参照できません。 **READWRITE** を指定した場合、どの I/O ステートメントもこの接続を参照できます。 **ACTION=** 指定子を省略した場合、デフォルト値は実際のファイル許可により決まります。

- **STATUS=** 指定子の値が **OLD** または **UNKNOWN** であり、ファイルがすでに存在している場合、
  - **READWRITE** を指定した状態でファイルがオープンされます。
  - 上記の状態が発生しえない場合は、**READ** を指定した状態でファイルがオープンされます。
  - 上記の 2 つの状態がともに発生しえない場合は、**WRITE** を指定した状態でファイルがオープンされます。
- **STATUS=** 指定子の値が **NEW**、**REPLACE**、**SCRATCH**、または **UNKNOWN** であり、ファイルがまだ存在していない場合:
  - **READWRITE** を指定した状態でファイルがオープンされます。
  - 上記の状態が発生しえない場合は、**WRITE** を指定した状態でファイルがオープンされます。

DELIM= *char_expr*
区切り文字がある場合に、リスト指示または名前リスト形式設定により書き込まれた文字定数を区切るために、どのような区切り文字を使用するかを指定します。 *char_expr* は、スカラー文字式で、その値は、**APOSTROPHE**、**QUOTE**、または **NONE** とならなければなりません。 **APOSTROPHE** を指定した場合、アポストロフィによって文字定数が区切られます。文字定数の中のすべての アポストロフィは 2 個指定されます。 **QUOTE** を指定した場合、二重引用符によって文字定数が区切られます。文字定数の中の二重引用符はすべて 2 個指定されます。 **NONE** を指定した場合、文字定数は区切られません。また、文字定数の中のどの文字も 2 個指定される必要はありません。デフォルト値は **NONE** です。 **DELIM=** 指定子は定様式 I/O 用に接続されるファイルにのみ有効です。ただし、定様式レコードの入力時には無視されます。

PAD= *char_expr*
入力レコードをブランクで埋め込むかどうかを指定します。*char_expr* はスカラー文字式で、式の値は、**YES** または **NO** のいずれかになります。 **YES** を指定した場合、入力リストが指定され、かつ、定様式仕様でレコードに含まれるものよりも多くのデータが必要なときは、定様式入力レコードはブランクで埋め込まれます。 **NO** を指定した場合、入力リストと定様式仕様に対してレコードに含まれるものよりも多くの文字を与えることはできません。デフォルト値は **YES** です。 **PAD=** 指定子は定様式 I/O 用に接続されるファイルにのみ有効です。ただし、定様式レコードの出力時には無視されます。

IBM 拡張

-qxlf77 コンパイラー・オプションに **noblankpad** サブオプションを指定し、ファイルが定様式直接 I/O 用に接続される場合、**PAD=** 指定子を省略すると、デフォルト値は **NO** になります。

IBM 拡張 の終り

規則

装置が既存のファイルに接続している場合、その装置に対して **OPEN** ステートメントを実行することができます。 **OPEN** ステートメントに **FILE=** 指定子を指定しない場合、その装置に接続されるファイルは、その装置が現在接続されているファイルと同じものになります。

装置に接続したいファイルが、その装置が現在接続されているファイルと異なる場合は、**OPEN** ステートメントの実行の直前に、**STATUS=** 指定子を指定せずに **CLOSE** ステートメントをその装置に対して実行した場合と同じ結果になります。

装置に接続したいファイルが、その装置が現在接続されているファイルと同じである場合は、現在有効な値と異なる値をとることのできる指定子は、**BLANK=**、**DELIM=**、

PAD=、**ERR=**、および **IOSTAT=** だけです。 **OPEN** ステートメントを実行すると、**BLANK=**、**DELIM=**、または **PAD=** 指定子の新しい値が有効になります。しかし、指定しない指定子およびファイルの位置は何の影響も受けません。事前に実行された **OPEN** の **ERR=** と **IOSTAT=** 指定子は、現在の **OPEN** ステートメントには効果がありません。 **STATUS=** 指定子を指定する場合、値 **OLD** でなければなりません。現在装置に接続されているファイルと同一のファイルを指定するには、同一のファイル名を指定するか、**FILE=** 指定子を省略するか、または同一のファイルにシンボリック・リンクされているファイルを指定します。

ファイルが装置に接続されている場合、そのファイルおよび異なる装置に対する **OPEN** ステートメントは実行されません。

IBM 拡張

STATUS= 指定子の値が **OLD**、 **NEW**、または **REPLACE** の場合、**FILE=** 指定子はオプションです。

事前接続ファイルと標準エラー・デバイス以外のファイルに装置 0 を接続することはできません。しかし、**BLANK=**、**DELIM=**、および **PAD=** 指定子の値を変更することはできます。

IBM 拡張 の終り

ERR= と **IOSTAT=** 指定子が設定されているときにエラーが検出されると、**ERR=** 指定子によって指定されたステートメントに対して転送が行われ、正の整数値が *ios* に割り当てられます。

IBM 拡張

IOSTAT= も **ERR=** も指定していない場合は、以下のとおりです。
- 重大なエラーが検出されるとプログラムは停止します。
- 回復可能エラーが検出され、 **ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、プログラムは次のステートメントの処理を継続します。オプションが **NO** に設定されていると、プログラムは停止します。

IBM 拡張 の終り

例

```
!   Open a new file with name fname

CHARACTER*20 FNAME
FNAME = 'INPUT.DAT'
OPEN(UNIT=8,FILE=FNAME,STATUS='NEW',FORM='FORMATTED')
```

```
OPEN (4,FILE="myfile")
OPEN (4,FILE="myfile", PAD="NO")  ! Changing PAD= value to NO

!   Connects unit 2 to a tape device for unformatted, sequential
!   write-only access:

OPEN (2, FILE="/dev/rmt0",ACTION="WRITE",POSITION="REWIND", &
&   FORM="UNFORMATTED",ACCESS="SEQUENTIAL",RECL=32767)
```

関連情報

- 222 ページの『装置の接続』
- 923 ページの『付録 A. 異なる標準の間の互換性』の項目 3
- 229 ページの『条件および IOSTAT 値』
- 「ユーザーズ・ガイド」の『*実行時オプションの設定*』
- 「ユーザーズ・ガイド」の『**-qposition** オプション』
- 「ユーザーズ・ガイド」の『**-qxlf77** オプション』
- 312 ページの『CLOSE』
- 445 ページの『READ』
- 500 ページの『WRITE』

OPTIONAL

目的

OPTIONAL 属性は、プロシージャーの参照時に、仮引き数を実引き数に関連付ける必要はないことを指定します。

構文

```
>>──OPTIONAL──┬──────┬──dummy_arg_name_list──────────────────────────><
              └─ :: ─┘
```

規則

オプションの仮引き数を指定したプロシージャーの参照では、明示インターフェースが必要です。

実引き数がオプションの仮引き数と関連付けられているかどうかを確認するには、**PRESENT** 組み込み関数を使用します。仮引き数が存在していることを確認しないでオプションの仮引き数を参照しないでください。

仮引き数が、実引き数と関連付けられている場合、サブプログラム内に存在しているものと見なされます。この実引き数は、それ自体が存在する仮引き数の場合もあります

(伝搬の例)。仮引き数のうちオプションでないものは、必ず指定しなければなりません。つまり、オプションではない仮引き数は実引き数と関連付けなければなりません。

オプションの仮引き数のうち指定されていないものは、オプションの仮引き数に対応する実引き数として使用することができます。そのとき当然、このオプションの仮引き数は実引き数と関連付けられていないものと考えられます。オプションの仮引き数のうち指定されていないものには、以下の制約事項が適用されます。
- 仮データ・オブジェクトまたはサブオブジェクトの場合、定義または参照はできません。
- 仮プロシージャーの場合、参照はできません。
- オプションではない仮引き数に対応する実引き数として指定することはできません。ただし、**PRESENT** 組み込み関数の引き数として指定することはできます。
- 配列の場合、実引き数としてエレメント型プロシージャーに提供することはできません。ただし、同じランクの配列が実引き数として提供される場合はこの限りではありません。

定義演算子または定義割り当て用の明示インターフェースを指定するインターフェース本体内の仮引き数に **OPTIONAL** 属性を指定することはできません。

OPTIONAL 属性と互換性のある属性

- ALLOCATABLE
- DIMENSION
- EXTERNAL
- INTENT
- POINTER
- TARGET
- VALUE
- VOLATILE

例

```
SUBROUTINE SUB (X,Y)
  INTERFACE
    SUBROUTINE SUB2 (A,B)
      OPTIONAL :: B
    END SUBROUTINE
  END INTERFACE
  OPTIONAL :: Y
  IF (PRESENT(Y)) THEN          ! Reference to Y conditional
    X = X + Y                   ! on its presence
  ENDIF
  CALL SUB2(X,Y)
END SUBROUTINE

SUBROUTINE SUB2 (A,B)
  OPTIONAL :: B                 ! B and Y are argument associated,
  IF (PRESENT(B)) THEN          ! even if Y is not present, in
    B = B * A                   ! which case, B is also not present
```

```
      PRINT*, B
    ELSE
      A = A**2
      PRINT*, A
    ENDIF
  END SUBROUTINE
```

関連情報

- 200 ページの『オプションの仮引き数』
- 170 ページの『インターフェースの概念』
- 647 ページの『PRESENT (A)』
- 195 ページの『仮引き数』

PARAMETER

目的

PARAMETER 属性は定数の名前を指定します。

構文

```
                     ┌─,──────────────────────────────┐
                     ▼                                │
►►──PARAMETER──(────constant_name── = ──init_expr────┴──)──────────────────────────►◄
```

init_expr
    初期化式です。

規則

名前付き定数は、タイプ、形状、パラメーターを、同じ有効範囲単位内の仕様ステートメントで事前に指定するか、暗黙に宣言されていなければなりません。名前付き定数が暗黙的にタイプ付けされている場合、後続のタイプ宣言ステートメントまたは属性指定ステートメントにその定数が現れたときは暗黙のタイプとパラメーターの値を確認しなければなりません。

constant_name は 1 つの有効範囲単位内の 1 つの **PARAMETER** 属性に 1 度だけ定義することができます。

初期化式の中で指定されている名前付き定数は事前に定義しておく (直前のステートメントで定義できない場合は同一の **PARAMETER** の中か、あるいはタイプ宣言ステートメントの中で定義する) か、または使用関連付けかホスト関連付けを介してアクセス可能にしておかなければなりません。

初期化は、組み込み割り当ての規則を使用して名前付き定数に割り当てられます。名前付き定数が文字タイプで、その長さが継承される場合、初期化式の長さを採用します。

PARAMETER 属性と互換性のある属性

- DIMENSION
- PRIVATE
- PUBLIC

例

```
REAL, PARAMETER :: TWO=2.0

COMPLEX      XCONST
REAL         RPART,IPART
PARAMETER    (RPART=1.1,IPART=2.2)
PARAMETER    (XCONST = (RPART,IPART+3.3))

CHARACTER*2, PARAMETER :: BB='   '

   :
END
```

関連情報

- 110 ページの『初期化式』
- 28 ページの『データ・オブジェクト』

PAUSE

目的

PAUSE ステートメントは、一時的にプログラムの実行を中止し、キーワード **PAUSE** および、文字定数または数字ストリングを指定している場合はそれを、装置 0 に出力します。

構文

```
>>─PAUSE─┬───────────────┬──────────────────────────────────────><
         ├─char_constant─┤
         └─digit_string──┘
```

char_constant
: スカラー文字定数です。この値はホレリス定数であってはなりません。

digit_string
1 〜 5 桁からなるストリングです。

規則

IBM 拡張

PAUSE ステートメントの実行後は、**Enter** キーを押すと処理が続行されます。端末に装置 5 が接続されていない場合、**PAUSE** ステートメントはプログラムの実行を中止しません。

IBM 拡張 の終り

Fortran 95

PAUSE ステートメントは Fortran 95 では削除されています。

Fortran 95 の終り

例

```
PAUSE 'Ensure backup tape is in tape drive'
PAUSE 10            ! Output:  PAUSE 10
```

関連情報

- 927 ページの『削除された機能』

POINTER (Fortran 90)

目的

POINTER 属性はオブジェクトをポインター変数として指定します。

ポインター という用語は Fortran 90 **POINTER** 属性を持つオブジェクトを示します。XL Fortran の以前のバージョンで **POINTER** ステートメントとして記述されていたものは、整数 **POINTER** ステートメントの項で詳しく説明します。現バージョンでは、これらのポインターは整数ポインター といいます。

構文

```
              ┌─,────────────────────────────────────────┐
              ▼                                          │
►►─POINTER─┬────┬─object_name─┬──────────────────────────────┬─┴──────────►◄
           └─::─┘             └─(─deferred_shape_spec_list─)─┘
```

deferred_shape_spec
コロン (:) です。ここで、各コロンは次元を表します。

規則

object_name はデータ・オブジェクトまたは関数結果を示します。 **DIMENSION** 属性を指定して有効範囲単位内以外で *object_name* を宣言した場合、配列仕様は *deferred_shape_spec_list* でなければなりません。

object_name を整数 **POINTER**、**NAMELIST**、または **EQUIVALENCE** ステートメントに指定することはできません。 *object_name* が派生型定義のコンポーネントである場合、派生型で宣言された変数を **EQUIVALENCE**、**DATA**、または **NAMELIST** ステートメントで指定することはできません。

ポインター変数は、共通ブロックおよびブロック・データ・プログラム単位に指定することができます。

IBM 拡張

Fortran 90 ポインターを確実にスレッド固有のものとするためには、そのポインターに **SAVE** または **STATIC** 属性を指定しないでください。 これらの属性は、ユーザーによって明示的に指定されるか、または **-qsave** コンパイラー・オプションを使用することにより暗黙的に指定されます。ただし、静的でないポインターが、ターゲットが静的であるポインター代入ステートメント内で使用される場合、ポインターへのすべての参照は、実際は静的な共用ターゲットへの参照になることに注意してください。

IBM 拡張 の終り

POINTER 属性を持つコンポーネントを含むオブジェクトはそれ自体、 **TARGET**、**INTENT**、または **ALLOCATABLE** 属性を持つことができます。ただし、これをデータ転送ステートメントに指定することはできません。

POINTER 属性と互換性のある属性

- AUTOMATIC
- DIMENSION
- INTENT
- OPTIONAL
- PRIVATE
- PROTECTED
- PUBLIC
- SAVE
- STATIC
- VOLATILE

これらの属性はポインターに対してだけ有効であり、関連するターゲットに対しては有効ではありません。ただし、**DIMENSION** 属性だけは関連するターゲットに対して有効になります。

例

例:

```
INTEGER, POINTER :: PTR(:)
INTEGER, TARGET :: TARG(5)
PTR => TARG                  ! PTR is associated with TARG and is
                             !   assigned an array specification of (5)

PTR(1) = 5                   ! TARG(1) has value of 5
PRINT *, FUNC()
CONTAINS
  REAL FUNCTION FUNC()
    POINTER :: FUNC          ! Function result is a pointer

   ⋮
  END FUNCTION
END
```

IBM 拡張

例 2: Fortran 90 ポインターとスレッド・セーフ

```
FUNCTION MYFUNC(ARG)                ! MYPTR is thread-specific.
INTEGER, POINTER :: MYPTR           !  every thread that invokes
                                    !  'MYFUNC' will allocate a
ALLOCATE(MYPTR)                     !  new piece of storage that
MYPTR = ARG                         !  is only accessible within
   ⋮
                                    !  that thread.
ANYVAR = MYPTR
END FUNCTION
```

IBM 拡張 の終り

関連情報

- 142 ページの『ポインターの割り当て』
- 475 ページの『TARGET』
- 553 ページの『ALLOCATED(ARRAY) または ALLOCATED(SCALAR)』
- 331 ページの『DEALLOCATE』
- 167 ページの『ポインター関連付け』
- 90 ページの『据え置き形状配列』

POINTER (整数)

IBM 拡張

目的

整数 **POINTER** ステートメントによって、変数 *int_pointer* の値を *pointee* の参照用アドレスとして使用することを指定できます。

Fortran 90 **POINTER** ステートメントと区別するために、このステートメントの名前は **POINTER** から整数 **POINTER** に変更されました。変更されたものは名前だけであり、ステートメントの機能と構文は XL Fortran の前のリリースと同じです。

構文

```
                ┌─,──────────────────────────────┐
                ▼                                │
►►──POINTER───(──int_pointer──,──pointee──)──┴──────────────────────────►◄
```

int_pointer
整数ポインター変数の名前です。

pointee 変数名、あるいは配列宣言子です。

規則

コンパイラーはポインティング先にストレージを割り振りません。ストレージは、実行時に、ポインターのストレージ・ブロック・アドレスを割り当てることによって、ポインティング先に関連付けられます。ポインティング先は静的または動的ストレージのいずれにも関連付けることができます。ポインティング先を参照するには、関連付けられるポインターが定義済みでなければなりません。

整数ポインターは、32 ビット・モードでは **INTEGER(4)** タイプおよび 64 ビット・モードでは **INTEGER(8)** タイプのスカラー変数であり、ユーザーが明示的にタイプを割り当てることはできません。整数ポインターと同じタイプの変数を使用できる式またはステートメントで、整数ポインターを使用することができます。ポインティング先には任意のデータ型を割り当てることができますが、ストレージ・クラスや初期値を割り当てることはできません。

POINTER ステートメントの中でポインティング先として指定された実際の配列をポインティング先配列といいます。ポインティング先配列の次元は、タイプ宣言ステートメント、 **DIMENSION** ステートメント、または整数 **POINTER** ステートメント自体の中で指定することができます。

-qddim コンパイラー・オプションを指定した場合は、メインプログラム内に現れるポインティング先配列には、調整可能配列仕様も指定できます。メインプログラムでもサブプログラムでも、次元のサイズはポインティング先が参照されるときに計算されます (動的に次元指定)。

-qddim コンパイラー・オプションを指定しなかった場合は、サブプログラム内に現れるポインティング先配列には調整可能配列仕様を指定でき、次元サイズは、ポインティング先の評価時ではなくサブプログラムに入った時点で計算されます。

ポインティング先と整数ポインターの定義および使用に関しては、次の制約事項が適用されます。
- ポインティング先のサイズをゼロにすることはできません。
- ポインティング先を、スカラー配列、想定形状配列、または明示的形状配列にできます。
- ポインティング先を **COMMON**、**DATA**、**NAMELIST**、または **EQUIVALENCE** ステートメントに指定することはできません。
- ポインティング先は次の属性を持つことはできません。 **EXTERNAL**、**ALLOCATABLE**、**POINTER**、**TARGET**、**INTRINSIC**、**INTENT**、**OPTIONAL**、**SAVE**、**STATIC**、**AUTOMATIC**、または **PARAMETER**。
- ポインティング先は仮引き数にはできません。したがって、**FUNCTION**、**SUBROUTINE**、または **ENTRY** ステートメントに指定することはできません。
- ポインティング先は自動オブジェクトにはできませんが、非定数境界または非定数長を持つことはできます。
- ポインティング先をインターフェース・ブロックの総称名として指定することはできません。
- 派生型のポインティング先は順序派生型でなければなりません。
- 関数値はポインティング先として使用できません。
- 整数ポインターを他のポインターのポインターとして使用することはできません。(ポインターはポインティング先にはなれません)。
- 整数ポインターが以下の属性を持つことはできません。
  - **ALLOCATABLE**

  - **DIMENSION**
  - **EXTERNAL**
  - **INTRINSIC**
  - **PARAMETER**
  - **POINTER**
  - **TARGET**
- 整数ポインターを **NAMELIST** のグループ名として指定することはできません。
- 整数ポインターをプロシージャーにすることはできません。

例

```
INTEGER A,B
POINTER (P,I)
IF (A<>0) THEN
  P=LOC(A)
ELSE
  P=LOC(B)
ENDIF
I=0         ! Assigns 0 to either A or B, depending on A's value
END
```

関連情報

- 168 ページの『整数ポインター関連付け』
- 617 ページの『LOC (X)』
- 「ユーザーズ・ガイド」の『**-qddim オプション**』

IBM 拡張 の終り

PRINT

目的

PRINT ステートメントはデータ転送出力ステートメントです。

構文

```
>>-PRINT--+-name-------------------------+--------------------------><
          '-format--+------------------+-'
                    '-,-output_item_list-'
```

name 名前リストのグループ名です。

output_item

出力リスト項目です。出力リストには転送するデータを指定します。出力リスト項目には、次のものを指定できます。

- 変数。この配列は、すべての配列エレメントが、ストレージに並んでいる順序で指定されているかのように処理されます。

  ポインターはターゲットと関連付ける必要があり、割り振り可能オブジェクトは割り振る必要があります。派生型のオブジェクトは、このステートメントの有効範囲単位の外側にある最終コンポーネントを持つことはできません。 *output_item* を評価しても、ポインターを含む派生型のオブジェクトは発生しません。 定様式ステートメント内の構造体のコンポーネントは、派生型定義で現れる順序で指定されているかのように処理されます。不定様式ステートメントでは、構造体コンポーネントは内部表示の 1 つの値として処理されます (埋め込みを含みます)。
- 式
- 暗黙 **DO** リスト。詳細は 438ページに記載されています。

format 出力操作で使用する形式を指定する形式指定子です。 *format* は形式識別子で、次のいずれかです。

- **FORMAT** ステートメントのステートメント・ラベル。 **FORMAT** ステートメントは同じ有効範囲単位内になければなりません。
- スカラー **INTEGER(4)** または **INTEGER(8)** 変数の名前。これには **FORMAT** ステートメントのステートメント・ラベルが割り当てられています。 **FORMAT** ステートメントは同じ有効範囲単位内になければなりません。

Fortran 95

Fortran 95 ではステートメント・ラベルの割り当ては行えません。

Fortran 95 の終り

- 文字定数。これはホレリス定数であってはなりません。これは左括弧で始まり、右括弧で終わっていなければなりません。両括弧の間で使用できるのは、**FORMAT** ステートメントに記述した形式コードだけです。ブランク文字は左括弧の前または右括弧の後ろにあってもかまいません。
- 左端の文字位置の部分が有効な形式になっている文字データを含む文字変数。有効な形式とは左括弧で始まり、右括弧で終わる形式です。両括弧の間で使用できるのは、 372 ページの『FORMAT』に記述した形式コードだけです。ブランク文字は左括弧の前または右括弧の後ろにあってもかまいません。
- 非文字組み込みタイプの配列。
- 文字式。ただし、オペランドが定数の名前でない場合、長さの継承を指定するオペランドの連結を含む文字式を除きます。
- リスト指示形式設定を指定するアスタリスク。

- 事前に定義した名前リストのリスト名を指定する名前リスト指定子。

-qport=typestmt コンパイラー・オプションを指定すると、 **PRINT** ステートメントと同一の機能を持つ **TYPE** ステートメントを使用できるようになります。

暗黙 DO リスト

```
►►─(─do_object_list─ , ─do_variable = arith_expr1, arith_expr2──────────────►

►─┬───┬──┬─────────────┬─)─────────────────────────────────────────────────►◄
  └─,─┘  └─arith_expr3─┘
```

do_object
: 出力リスト項目です。

do_variable
: 整数、または実数タイプのスカラー変数です。

arith_expr1、*arith_expr2*、および *arith_expr3*
: スカラー数式です。

暗黙 **DO** リストの範囲は *do_object_list* です。繰り返し回数および **DO** 変数の値は、**DO** ステートメントの場合と同様に *arith_expr1*、*arith_expr2*、および *arith_expr3* で決まります。暗黙 **DO** リストが実行されると、暗黙 **DO** リストの繰り返しごとに、*do_object_list* 内の項目が 1 つ指定され、**DO** 変数のその時点の値に応じた適切な値に置き換えられます。

例

```
   PRINT 10, A,B,C
10 FORMAT (E4.2,G3.2E1,B3)
```

関連情報

- 217 ページの『第 8 章 I/O の概念』
- 239 ページの『第 9 章 I/O の形式設定』
- **-qport=typestmt** について詳しくは、「ユーザーズ・ガイド」を参照してください。
- 927 ページの『削除された機能』

PRIVATE

目的

PRIVATE 属性は、使用関連付けを介してもモジュール外ではモジュール・エンティティーにアクセスできないことを指定します。

構文

```
>>─PRIVATE─┬────────────────────┬──────────────────────────────────><
           └─┬────┬─access_id_list─┘
             └─::─┘
```

access_id

総称仕様、または変数、プロシージャー、派生型、定数、名前リスト・グループの名前です。

規則

PRIVATE 属性はモジュールの有効範囲にだけ指定できます。

1 つのモジュールに複数の **PRIVATE** ステートメントを指定できますが、*access_id_list* を省略できるステートメントは 1 つだけです。 *access_id_list* を指定していない **PRIVATE** ステートメントでは、モジュール内で潜在的にアクセス可能なエンティティーのデフォルトのアクセス可能度をプライベートに設定しています。このようなステートメントを含むモジュールに *access_id_list* を持たない **PUBLIC** ステートメントを指定することはできません。モジュールにこのようなステートメントを指定していない場合、デフォルトのアクセス可能度はパブリックです。明示的にアクセス可能度を指定していないエンティティーにはデフォルトにアクセス可能度があります。

パブリックな総称識別子を持つプロシージャーの場合、特定の識別子がプライベートであったとしても、総称識別子でそのプロシージャーにアクセスできます。プライベートなアクセス可能度を持つプライベート仮引き数または関数結果がモジュール・プロシージャーに含まれる場合、そのモジュール・プロシージャーはプライベートなアクセス可能度を持つということを宣言しなければなりません。また、パブリックなアクセス可能度を持つ総称識別子をそのモジュール・プロシージャーに含むことはできません。

派生型定義の中で **PRIVATE** ステートメントを指定した場合、派生型のすべてのコンポーネントがプライベートになります。

派生型がプライベートである構造体はプライベートでなければなりません。プライベートなオブジェクトまたはプライベート・コンポーネントを含む名前リスト・グループはプライベートでなければなりません。プライベートな派生型のコンポーネントを持つ派生型はプライベートでなければなりません。あるいはプライベート・コンポーネントを持たなければなりません。任意の引き数がプライベートな派生型である場合、サブプログラムはプライベートでなければなりません。結果変数がプライベートな派生型である場合、関数はプライベートでなければなりません。

PRIVATE 属性と互換性のある属性

• ALLOCATABLE	• PARAMETER	• STATIC
• DIMENSION	• POINTER	• TARGET
• EXTERNAL	• PROTECTED	• VOLATILE
• INTRINSIC	• SAVE	

例

```
MODULE MC
   PUBLIC                     ! Default accessibility declared as public
   INTERFACE GEN
      MODULE PROCEDURE SUB1, SUB2
   END INTERFACE
   PRIVATE SUB1               ! SUB1 declared as private
   CONTAINS
      SUBROUTINE SUB1(I)
         INTEGER I
         I = I + 1
      END SUBROUTINE SUB1
      SUBROUTINE SUB2(I,J)
         I = I + J
      END SUBROUTINE
END MODULE MC

PROGRAM ABC
   USE MC
   K = 5
   CALL GEN(K)                ! SUB1 referenced because GEN has public
                              ! accessibility and appropriate argument
                              ! is passed
   CALL SUB2(K,4)
   PRINT *, K                 ! Value printed is 10
END PROGRAM
```

関連情報

- 41 ページの『派生型』
- 183 ページの『モジュール』
- 442 ページの『PROTECTED』
- 444 ページの『PUBLIC』

PROGRAM

目的

PROGRAM ステートメントは、そのプログラム単位がメインプログラムであることを示します。メインプログラムとは、実行時に実行可能プログラムを呼び出したときにシステムから制御を受け取るプログラム単位のことです。

構文

```
►►──PROGRAM──name──────────────────────────────────────────────────►◄
```

name このステートメントが指定されているメインプログラムの名前です。

規則

PROGRAM ステートメントはオプションです。

PROGRAM ステートメントを指定する場合は、メインプログラムの先頭のステートメントでなければなりません。

対応する **END** ステートメントの中にプログラム名を指定する場合、その名前は *name* に一致しなければなりません。

プログラム名は実行可能プログラムに対してグローバルです。この名前には、実行可能プログラム内の共通ブロック、外部プロシージャー、または他のプログラム単位のいずれかと同じ名前を付けることはできません。また、メインプログラムに対してローカルな名前を付けることはできません。

名前はタイプを持ちません。また、どのタイプ宣言にもどの仕様ステートメントにも指定することはできません。サブプログラムまたはメインプログラム自身からメインプログラムを参照することはできません。

例

```
PROGRAM DISPLAY_NUMBER_2
   INTEGER A
   A = 2
   PRINT *, A
END PROGRAM DISPLAY_NUMBER_2
```

関連情報

181 ページの『メインプログラム』

PROTECTED

IBM 拡張

目的

PROTECTED 属性によって、モジュール・エンティティーの変更をより大きく制御できます。モジュール・プロシージャーが、保護モジュール・エンティティーまたはそのサブオブジェクトを変更できるのは、同じモジュールがプロシージャーとエンティティーの両方に定義されている場合のみです。

構文

PROTECTED 属性は、モジュールの仕様部分にしか指定できません。

```
>>──PROTECTED──┬────┬──entity_declaration_list──────────────────────────><
               └─::─┘
```

entity 共通ブロックにない名前付き変数

規則

EQUIVALENCE ステートメントによって宣言されたオブジェクトが **PROTECTED** 属性を持つように指定した場合、その **EQUIVALENCE** ステートメントで指定されたオブジェクトはすべて **PROTECTED** 属性を持たなければなりません。

使用関連付けを介してアクセスされる、**PROTECTED** 属性を持つ非ポインター・オブジェクトは定義できません。

PROTECTED 属性を整数ポインターに指定してはなりません。

使用関連付けを介してアクセスされる、**PROTECTED** 属性を持つポインター・オブジェクトを以下で指定してはなりません

- **NULLIFY** ステートメントまたは **POINTER** 代入ステートメントのポインター・オブジェクトとして
- **ALLOCATE** または **DEALLOCATE** ステートメントの割り振り可能オブジェクトとして
- 関連する仮引き数が **INTENT(INOUT)** または **INTENT(OUT)** 属性を持つポインターである場合の、プロシージャー参照の実引き数として

PROTECTED 属性と互換性のある属性

- ALLOCATABLE
- AUTOMATIC
- DIMENSION
- INTENT
- OPTIONAL
- POINTER
- PRIVATE
- PUBLIC
- SAVE
- STATIC
- TARGET
- VOLATILE

例

次の例で、*age* と *val* の両方の値は、これらが宣言されているサブルーチンによってのみ変更できます。

```
module mod1
    integer, protected :: val
    integer :: age
    protected :: age
    contains
        subroutine set_val(arg)
            integer arg
             val = arg
        end subroutine
         subroutine set_age(arg)
            integer arg
             age  = arg
        end subroutine
end module
program dt_init01
    use mod1
    implicit none
    integer :: value, his_age
    call set_val(88)
    call set_age(38)
    value = val
    his_age = age
    print *, value, his_age
end program
```

関連情報

183 ページの『モジュール』

438 ページの『PRIVATE』

444 ページの『PUBLIC』

IBM 拡張 の終り

PUBLIC

目的

PUBLIC 属性は、他のプログラム単位が使用関連付けを介してモジュール・エンティティーにアクセスできることを指定します。

構文

```
►►──PUBLIC──┬────────────────────────────┬─────────────────────────────────►◄
            └──┬────┬──access_id_list──┘
               └─::─┘
```

access_id
: 総称仕様、または変数、プロシージャー、派生型、定数、名前リスト・グループの名前です。

規則

PUBLIC 属性はモジュールの有効範囲にだけ指定できます。

1 つのモジュールに複数の **PUBLIC** ステートメントを指定できますが、*access_id_list* を省略できるステートメントは 1 つだけです。 *access_id_list* を指定していない **PUBLIC** ステートメントでは、モジュール内で潜在的にアクセス可能なエンティティーのデフォルトのアクセス可能度をパブリックに設定しています。このようなステートメントを含むモジュールに *access_id_list* を持たない **PRIVATE** ステートメントを指定することはできません。モジュールにこのようなステートメントを指定していない場合、デフォルトのアクセス可能度はパブリックです。明示的にアクセス可能度を指定していないエンティティーにはデフォルトにアクセス可能度があります。

パブリックな総称識別子を持つプロシージャーの場合、特定の識別子がプライベートであったとしても、総称識別子でそのプロシージャーにアクセスできます。プライベートなアクセス可能度を持つプライベート仮引き数または関数結果がモジュール・プロシージャーに含まれる場合、そのモジュール・プロシージャーはプライベートなアクセス可能度を持つということを宣言しなければなりません。また、パブリックなアクセス可能度を持つ総称識別子をそのモジュール・プロシージャーに含むことはできません。

IBM 拡張

パブリックというアクセス可能度を持つエンティティーは **STATIC** 属性を持つことはできませんが、モジュール内のパブリック・エンティティーはモジュール内の **IMPLICIT STATIC** ステートメントによって影響されません。

IBM 拡張 の終り

PUBLIC 属性と互換性のある属性

- ALLOCATABLE
- DIMENSION
- EXTERNAL
- INTRINSIC
- PARAMETER
- POINTER
- PROTECTED
- SAVE
- TARGET
- VOLATILE

例

```
MODULE MC
   PRIVATE                       ! Default accessibility declared as private
   PUBLIC GEN                    ! GEN declared as public
   INTERFACE GEN
      MODULE PROCEDURE SUB1
   END INTERFACE
   CONTAINS
      SUBROUTINE SUB1(I)
         INTEGER I
         I = I + 1
      END SUBROUTINE SUB1
END MODULE MC
PROGRAM ABC
   USE MC
   K = 5
   CALL GEN(K)                   ! SUB1 referenced because GEN has public
                                 !   accessibility and appropriate argument
                                 !   is passed
   PRINT *, K                    ! Value printed is 6
END PROGRAM
```

関連情報

- 438 ページの『PRIVATE』
- 183 ページの『モジュール』

READ

目的

READ ステートメントはデータ転送入力ステートメントです。

構文

```
►►─READ──┬─name─────────────────────────────────┬──────────────────────►◄
         ├─format──┬────────────────────┬───────┤
         │         └─,─input_item_list──┘       │
         └─(─io_control_list─)──┬─────────────────┬┘
                                └─input_item_list─┘
```

format **FMT=***format* の項で解説する形式識別子です。これはホレリス定数であってはなりません。

name 名前リストのグループ名です。

input_item

入力リスト項目です。入力リストには転送するデータを指定します。入力リスト項目には、次のものを指定できます。

- 変数名。ただし、想定サイズ配列を除きます。この配列は、すべての配列エレメントが、ストレージに並んでいる順序で指定されているかのように処理されます。

  ポインターは定義可能ターゲットと関連させる必要があり、割り振り可能オブジェクトは割り振る必要があります。派生型のオブジェクトは、このステートメントの有効範囲単位の外側にある最終コンポーネントを持つことはできません。*input_item* を評価しても、ポインターを含む派生型のオブジェクトは発生しません。 定様式ステートメント内の構造体のコンポーネントは、派生型定義で現れる順序で指定されているかのように処理されます。不定様式ステートメントでは、構造体コンポーネントは内部表示の 1 つの値として処理されます (埋め込みを含みます)。
- 暗黙 **DO** リスト。詳細は 451 ページの『暗黙 DO リスト』に記載されています。

io_control

装置指定子 (**UNIT=**) を必ず 1 つ含んでいなければならないリストです。このリストには、他の有効な指定子をそれぞれ 1 つずつ入れることができます。

[UNIT=] *u*

入力操作で使用する装置を指定する装置識別子です。 *u* は、外部装置識別子または内部ファイル識別子です。

IBM 拡張

外部装置識別子は外部ファイルを示します。それは次のうちの 1 つです。

- 値が 0 ～ 2,147,483,647 の範囲内にある整数式。
- アスタリスク、外部装置 5 を識別し、標準入力にあらかじめ接続されているもの。

IBM 拡張 の終り

内部ファイル識別子は内部ファイルを示します。これは、ベクトル添え字を持つ配列セクションにはならない文字変数の名前です。

オプションの文字である **UNIT=** を省略する場合、*u* は *io_control_list* の最初の項目でなければなりません。オプションの文字である **UNIT=** を指定する場合、オプションの文字 **FMT=** またはオプションの文字 **NML=** もなければなりません。

[FMT=] *format*

入力操作で使用する形式を指定する形式指定子です。 *format* は形式識別子で、次のいずれかです。

- **FORMAT** ステートメントのステートメント・ラベル。 **FORMAT** ステートメントは同じ有効範囲単位内になければなりません。
- スカラー **INTEGER(4)** または **INTEGER(8)** 変数の名前。これには **FORMAT** ステートメントのステートメント・ラベルが割り当てられています。 **FORMAT** ステートメントは同じ有効範囲単位内になければなりません。

Fortran 95

Fortran 95 ではステートメント・ラベルの割り当ては行えません。

Fortran 95 の終り

- 文字定数。これは左括弧で始まり、右括弧で終わっていなければなりません。両括弧の間で使用できるのは、**FORMAT** ステートメントに記述した形式コードだけです。ブランク文字は左括弧の前または右括弧の後ろにあってもかまいません。
- 左端の文字位置の部分が有効な形式になっている文字データを含む文字変数。有効な形式とは左括弧で始まり、右括弧で終わる形式です。両括弧の間で使用できるのは、 372 ページの『FORMAT』に記述した形式コードだけです。ブランク文字は左括弧の前または右括弧の後ろにあってもかまいません。 *format* が配列エレメントの場合、形式識別子の長さは配列エレメントの長さを超えてはなりません。
- 非文字組み込みタイプの配列。文字配列の項で説明したように、データは有効な形式識別子でなければなりません。
- 文字式。ただし、オペランドが定数の名前でない場合、長さの継承を指定するオペランドの連結を含む文字式を除きます。
- リスト指示形式設定を指定するアスタリスク。
- 事前に定義した 名前リストのリスト名を指定する名前リスト指定子。

オプションの文字である **FMT=** を省略する場合、*io_control_list* 内の 2 番目の項目は *format* でなければなりません。最初の項目は、オプションの文字

UNIT= を省略した装置指定子でなければなりません。 1 つの入力ステートメントに **NML=** と **FMT=** の両方を指定することはできません。

POS=*integer_expr*

integer_expr はスカラー整数式で、式の値は 0 より大きくなければなりません。**POS=** はストリーム・アクセス用に接続されたファイル内で読み取られるファイル記憶単位のファイル位置を示します。 **POS=** は、位置決めを行うことができないファイルに使用してはなりません。

REC= *integer_expr*

直接アクセス用に接続されたファイルの中で読み取るレコードの番号を指定するレコード指定子です。 **REC=** 指定子を使用できるのは、直接入力の場合に限られます。 *integer_expr* は正の値を持つ整数式です。リスト指示または名前リスト形式設定を使用している場合、および装置指定子で内部ファイルを使用している場合、レコード指定子は有効ではありません。 **END=** 指定子を同時に用いることはできません。レコード指定子は、ファイル内のレコードの相対的な位置を示します。最初のレコードの相対位置番号は 1 です。ストリーム・アクセス用に接続された装置を指定するデータ転送ステートメントで **REC=** を指定してはなりません。また、**POS=** 指定子を使用してはなりません。

IBM 拡張

ID= *integer_variable*

データ転送が非同期に行われることを示します。 *integer_variable* は、**INTEGER(4)** またはデフォルト整数のタイプのスカラーです。エラーが検出されない場合、*integer_variable* は、非同期データ転送ステートメントの実行後に 1 つの値で定義されます。この値は、対応する **WAIT** ステートメント内で使用されなければなりません。

非同期データ転送は、直接不定様式、順次不定様式、ストリーム不定様式のいずれかでなければなりません。内部ファイルへの非同期 I/O は禁止されています。ロー文字装置への非同期 I/O (たとえば、テープまたはロー論理ボリュームへの非同期 I/O) は、禁止されています。 *integer_variable* を、データ転送 I/O リストのエンティティーや、データ転送 I/O リストの *io_implied_do* の *do_variable* に関連させることはできません。 *integer_variable* が配列エレメント参照の場合、その添え字値は、データ転送、*io_implied_do* 処理、または *io_control_spec* 内の他の指定子の定義や評価などによって影響を受けてはなりません。

IBM 拡張 の終り

IOSTAT= *ios*

I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** またはデフ

ォルト整数のタイプの変数です。 **IOSTAT=** 指定子は、エラー・メッセージを抑制します。ステートメントの実行が完了すると、*ios* の値は次のように定義されます。

- エラーが発生しなかった場合、ファイルの終わりが検出されなかった場合、レコードの終わりが検出されなかった場合は、ゼロが定義されます。
- エラーが発生した場合は正の値が定義されます。
- ファイルの終わりが検出されていて、しかも、エラーが発生しなかった場合は、負の値が定義されます。
- レコードの終わりが検出されて、しかも、エラーが発生しなかったかファイルの終わりが検出された場合は、ファイルの終わりの値とは異なる負の値が定義されます。

ERR= *stmt_label*

エラーが発生した場合に制御が移される実行可能ステートメントのラベルを指定するエラー指定子です。 **ERR=** 指定子をコーディングすると、エラー・メッセージは抑制されます。

END= *stmt_label*

エラーが発生せずにファイル終了レコードまで達した場合に、プログラムの実行を継続するステートメント・ラベルを指定するファイルの終わり指定子です。外部ファイルはファイルの最終レコードの後に位置付けられます。**IOSTAT=** 指定子を指定した場合、この指定子には負の値が割り当てられます。 **NUM=** 指定子を指定した場合、この指定子には整数値が割り当てられます。エラーが発生した場合、そのステートメントに **SIZE=** 指定子が含まれていると、指定した変数は整数値で定義されます。**END=** 指定子をコーディングすると、ファイルの終わりに関するエラー・メッセージが抑止されます。この指定子は、順次アクセスまたは直接アクセス用に接続された装置に指定することができます。

IBM 拡張

NUM= *integer_variable*

入出力リストとファイルの間で転送されるデータのバイト数を指定する数指定子です。 *integer_variable* は、**INTEGER(4)** タイプ、64 ビットの **INTEGER(8)** タイプ、またはデフォルトの整数タイプのスカラー変数名です。 **NUM=** 指定子を使用できるのは、不定様式出力の場合に限られます。**NUM** パラメーターをコーディングすると、出力リストに表示されるバイト数が、レコードに書き込めるバイト数よりも大きい場合、出されるエラー表示は抑止されます。この場合、*integer_variable* の値は、書き込み可能な最大レコード長に設定されます。残りの出力リスト項目からのデータは、これ以後のレコードには書き込まれません。

IBM 拡張 の終り

[NML=] *name*
事前に定義した名前リストの名前を指定する名前リスト指定子です。オプションの文字の **NML=** を指定しない場合、リストの 2 番目のパラメーターは名前リストの名前でなければなりません。また、最初の項目は **UNIT=** を省略した装置指定子でなければなりません。 **NML=** と **UNIT=** の両方を指定する場合、すべてのパラメーターを任意の順序で指定することができます。 **NML=** 指定子は **FMT=** の代替指定子です。1 つの入力ステートメントに **NML=** と **FMT=** の両方を指定することはできません。

ADVANCE= *char_expr*
このステートメントについて非事前入力が発生するかどうかを決定する事前指定子です。 *char_expr* はスカラー文字式で、式の値は、**YES** または **NO** のいずれかに評価されます。 **NO** を指定した場合、非事前入力が発生します。 **YES** を指定した場合、事前定様式順次入力、または事前ストリーム入力が発生します。デフォルト値は **YES** です。**ADVANCE=** を指定できるのは、内部ファイル単位指定子を指定しない明示的な形式仕様を持つ定様式の順次 **READ** ステートメントまたはストリーム **READ** ステートメント内だけです。

SIZE= *count*
現在の入力ステートメントの実行中にデータ編集記述子によって転送される文字数を決定する文字カウント指定子です。 *count* は、デフォルトの整数タイプ、 IBM **INTEGER(4)** タイプ、または 64 ビットの **INTEGER(8)** タイプのスカラー変数です。 IBM 埋め込みとして挿入されるブランクはカウントされません。

EOR= *stmt_label*
レコードの終わり指定子です。この指定子を指定し、レコードの終わりが検出され、ステートメントの実行中にエラーが発生しなかった場合は、以下のとおりです。 **PAD=** がある場合、次のようになります。

1. **PAD=** 指定子の値が **YES** の場合、レコードはブランクで埋め込まれ、入力リスト項目と、レコードが含む文字よりも多くの文字を必要とするデータ編集記述子を満たします。
2. **READ** ステートメントの実行が終了します。
3. **READ** ステートメントに指定されているファイルが現在のレコードの後ろに置かれます。
4. **IOSTAT=** 指定子を指定している場合、指定した変数はファイルの終わりの値と異なる負の値で定義されます。
5. **SIZE=** 指定子を指定している場合、指定した変数は整数値で定義されます。
6. **EOR=** 指定子によって指定されているステートメント・ラベルを含むステートメントの実行が継続されます。
7. レコード終わりに関するメッセージが抑止されます。

暗黙 DO リスト

```
▶▶─(─do_object_list─ , ─do_variable = arith_expr1, arith_expr2──────────────▶

▶─┬───┬──┬──────────────┬─)──────────────────────────────────────────────▶◀
  └─,─┘  └─arith_expr3──┘
```

do_object
: 出力リスト項目です。

do_variable
: 整数、または実数タイプのスカラー変数です。

arith_expr1、*arith_expr2*、および *arith_expr3*
: スカラー数式です。

暗黙 **DO** リストの範囲は *do_object_list* です。繰り返し回数および **DO** 変数の値は、**DO** ステートメントの場合と同様に *arith_expr1*、*arith_expr2*、および *arith_expr3* で決まります。暗黙 **DO** リストが実行されると、暗黙 **DO** リストの繰り返しごとに、*do_object_list* 内の項目が 1 つ指定され、**DO** 変数のその時点の値に応じた適切な値に置き換えられます。

DO 変数または関連するデータ項目を入力リスト項目として *do_object_list* に指定することはできません。ただし、暗黙 **DO** リストの外にある同じ **READ** ステートメント内では **DO** 変数または関連するデータ項目を読み取ることができます。

規則

ERR=、 **EOR=** および **END=** 指定子によって指定されるステートメント・ラベルは **READ** ステートメントと同じ有効範囲単位内にある分岐ターゲット・ステートメントを参照していなければなりません。

EOR= 指定子または **SIZE=** 指定子を用いた場合は、値 **NO** を持つ **ADVANCE=** 指定子も指定する必要があります。

IBM 拡張

NUM= 指定子を指定した場合は、形式指定子も名前リスト指定子も指定することはできません。

IBM 拡張 の終り

IOSTAT=、**SIZE=**、**NUM=** 指定子に指定された変数を、入力リスト項目、名前リストのリスト項目、および暗黙 **DO** リストの **DO** 変数に関連付けることはできません。このような指定子変数が配列エレメントの場合、データ転送、暗黙 **DO** 処理、または他の指定子の定義または評価が、その添え字値に影響を与えてはいけません。

io_control_list を指定していない **READ** ステートメントは、外部装置識別子がアスタリスクである *io_control_list* を指定した **READ** ステートメントと同じ装置を指定します。

ERR= と **IOSTAT=** 指定子が設定されているときにエラーが検出されると、**ERR=** 指定子によって指定されたステートメントに対して転送が行われ、正の整数値が *ios* に割り当てられます。

IBM 拡張

ERR= か **IOSTAT=** 指定子が設定され、非同期データ転送中にエラーが検出されると、対応する **WAIT** ステートメントの実行は要求されません。

END= か **IOSTAT=** 指定子が設定され、非同期データ転送中にファイルの終わり条件が検出されると、対応する **WAIT** ステートメントの実行は要求されません。

変換エラーが検出され、 **CNVERR** 実行時オプションが **NO** に設定されている場合、**IOSTAT=** は設定されますが、 **ERR=** には分岐しません。

IOSTAT= も **ERR=** も指定していない場合は、以下のとおりです。
- 重大なエラーが検出されるとプログラムは停止します。
- 回復可能エラーが検出され、 **ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、プログラムは次のステートメントの処理を継続します。オプションが **NO** に設定されていると、プログラムは停止します。
- **ERR_RECOVERY** 実行時オプションが **YES** に設定されている場合、変換エラーが検出されると、プログラムは次のステートメントの処理を継続します。 **CNVERR** 実行時オプションが **YES** に設定されている場合、変換エラーは回復可能エラーとして処理されます。一方、**CNVERR=NO** の場合、エラーは変換エラーとして処理されます。

IBM 拡張 の終り

例

```
INTEGER A(100)
CHARACTER*4 B
READ *, A(LBOUND(A,1):UBOUND(A,1))
READ (7,FMT='(A3)',ADVANCE='NO',EOR=100) B
   :
100 PRINT *, 'end of record reached'
END
```

関連情報

- 225 ページの『データ転送ステートメントの非同期の実行』
- 「ユーザーズ・ガイド」の『*XL Fortran I/O のインプリメンテーションの詳細*』
- 229 ページの『条件および IOSTAT 値』
- 500 ページの『WRITE』
- 496 ページの『WAIT』
- 217 ページの『第 8 章 I/O の概念』
- 「ユーザーズ・ガイド」の『*実行時オプションの設定*』
- 927 ページの『削除された機能』

REAL

目的

REAL タイプ宣言ステートメントは、実数タイプのオブジェクトと関数の長さと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>—REAL—┬──────────────┬──┬─::───────────────────┬─entity_decl_list——————————><
        └kind_selector─┘  └─,─attr_spec_list─::─┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE
AUTOMATIC
DIMENSION (*array_spec*)
EXTERNAL
INTENT (*intent_spec*)
INTRINSIC
OPTIONAL
PARAMETER
POINTER
PRIVATE
PUBLIC
SAVE
STATIC
TARGET
VOLATILE

kind_selector

```
►►──┬─(─┬──────────┬─int_initialization_expr─)─┬──────────────────────►◄
    │   └─KIND── = ┘                          │
    │                             (1)         │
    └─ * ─int_literal_constant────────────────┘
```

注:

1　IBM 拡張

IBM 拡張

実数エンティティーの長さ (4、8、16) を指定します。*int_literal_constant* には、kind 型付きパラメーターは指定できません。

IBM 拡張 の終り

attr_spec
: 特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
: **IN**、**OUT**、または **INOUT** のいずれかです。

::
: ダブル・コロン・セパレーターです。複数の属性を指定するとき、または = *initialization_expr* あるいは => **NULL()** を使用するときに必要になります。

array_spec
: 次元境界のリストです。

entity_decl

```
>>─a─┬─┬──────────────────┬─┬────────────────────┬─┬──────────────>
     │ │      (1)    (2)  │ └─(─array_spec─)─┘   │
     │ └─ * ────────len───┘                      │
     │                     (3)      (4)          │
     └─(─array_spec─)─ * ────────len─────────────┘

>─┬──────────────────────────────────┬────────────────────────────><
  │                          (5)     │
  └─┬─/─initial_value_list─/─────────┤
    ├─ = ─initialization_expr────────┤
    │              (6)               │
    └─ => ─NULL()────────────────────┘
```

注:

1 IBM 拡張

2 IBM 拡張

3 IBM 拡張

4 IBM 拡張

5 IBM 拡張

6 Fortran 95

a オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

len *kind_selector* に指定されている長さをオーバーライドします。kind 型付きパラメーターを指定することはできません。エンティティーの長さは、許容できる長さ指定の 1 つを表す整数のリテラル定数でなければなりません。

IBM 拡張 の終り

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティー

に初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するために、タイプ宣言ステートメントを使用することができます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトである場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。オブジェクトがブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、そのオブジェクトを初期化できます。

IBM 拡張

また、オブジェクトがモジュール内の名前付き共通ブロックにある場合も、そのオブジェクトを初期化できます。

IBM 拡張 の終り

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、*initialization_expr* F95 または **NULL()** F95 を指定した場合、変数は最初に定義されます。

Fortran 95

宣言するエンティティーが派生型コンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルト初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、 *a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている T または F がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

例

```
REAL(8), POINTER :: RPTR
REAL(8), TARGET  :: RTAR
```

関連情報

- 31 ページの『実数』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

RECORD

IBM 拡張

目的

RECORD ステートメントは特殊な形式のタイプ宣言ステートメントです。他のタイプ宣言ステートメントとは異なり、**RECORD** ステートメントで宣言されたエンティティーの属性はこのステートメント自体には指定できません。

構文

レコード・エンティティー宣言は、以下の構文に従って行います。

record_stmt:

```
            ┌─,──────────────────────────────────────────┐
            ▼                                            │
►►──RECORD────/ type_name /──┬────┬──record_obj_dcl_list─┴───────────────►◄
                             └─::─┘
```

record_obj_dcl:

```
►►──record_object_name──┬────────────────┬───────────────────────────────►◄
                        └─(-array_spec-)─┘
```

type_name は有効範囲単位内でアクセス可能な派生型の名前でなければなりません。

規則

RECORD ステートメント内でエンティティーを初期化することはできません。

record_stmt は、その直前にある *type_name* によって指定された派生型のエンティティーを宣言します。

RECORD キーワードは、**IMPLICIT** または **FUNCTION** ステートメントの *type_spec* としては指定できません。

例

以下の例では、派生型変数を宣言するために、**RECORD** ステートメントが使用されています。

```
STRUCTURE /S/
  INTEGER I
END STRUCTURE
STRUCTURE /DT/
  INTEGER I
END STRUCTURE
RECORD/DT/REC1,REC2,/S/REC3,REC4
```

関連情報

- レコード構造および派生型について詳しくは、 41 ページの『派生型』を参照してください。

IBM 拡張 の終り

RETURN

目的

RETURN ステートメントは次のことを行います。

- 関数サブプログラムでは、サブプログラムの実行を終了し、サブプログラムの参照元のステートメントに制御を戻します。関数の値は、参照元のプロシージャーで使用可能です。
- サブルーチン・サブプログラムでは、そのサブプログラムを終了し、プロシージャー参照の後にある最初の実行可能ステートメントに制御を戻すか、あるいは、選択戻り点が指定されている場合は、そこに制御を戻します。

IBM 拡張

- メインプログラムでは、実行可能プログラムの実行を終了します。

IBM 拡張 の終り

構文

```
>>--RETURN--+------------------+--------------------------------------><
            |              (1) |
            '-arith_expr-------'
```

注:

1 IBM 拡張

IBM 拡張

arith_expr
: スカラー整数、実数、または複素数式です。式の値が整数でない場合、使用前に、**INTEGER(4)** に変換されます。*arith_expr* はホレリス定数であってはなりません。

IBM 拡張 の終り

規則

arith_expr は、サブルーチン・サブプログラム内でのみ指定できます。これは、代替戻り点を指定します。 *m* が *arith_expr* の値で、「$1 \leq m \leq$ (**SUBROUTINE** または **ENTRY** ステートメント内のアスタリスク数)」であれば、仮引き数リスト内の *m* 番目のアスタリスクが選択されます。その後、**CALL** ステートメント内の *m* 番目の選択戻り指定子として指定されているステートメント・ラベルを持つ呼び出しプロシージャー内のステートメントに、制御が戻されます。たとえば、*m* の値が 5 の場合、**CALL** ス

テートメント内の 5 番目の選択戻り指定子として指定されているステートメント・ラベルを持つステートメントに制御が戻されます。

arith_expr を省略するか、あるいはその値 (*m*) が、1 から (**SUBROUTINE** または **ENTRY** ステートメント内のアスタリスクの数) までの範囲外にある場合、通常の戻りが実行されます。制御は、呼び出しプロシージャー内の **CALL** の次のステートメントに戻されます。

RETURN が実行されると、サブプログラムの仮引き数とそのサブプログラムのインスタンスに提供される実引き数の間の関連付けが終了します。サブプログラムにとってローカルであるエンティティーは、 74 ページの『未定義を発生させるイベント』に説明されているものを除いて、すべて未定義になります。

1 つのサブプログラムに複数の **RETURN** ステートメントを指定することもできますが、何も指定しなくてもかまいません。関数サブプログラムまたはサブルーチン・サブプログラム内の **END** ステートメントは、**RETURN** ステートメントと同じ働きをします。

例

```
CALL SUB(A,B)
CONTAINS
  SUBROUTINE SUB(A,B)
    INTEGER :: A,B
    IF (A.LT.B)
      RETURN             ! Control returns to the calling procedure
    ELSE
   ⋮
    END IF
  END SUBROUTINE
END
```

関連情報

- 206 ページの『仮引き数としてのアスタリスク』
- 選択戻り点の説明は、 193 ページの『実引き数の仕様』
- 74 ページの『未定義を発生させるイベント』

REWIND

目的

REWIND ステートメントは、順次アクセス用に接続された外部ファイルをファイルの最初のレコードの先頭に位置付けます。ストリーム・アクセスの場合、**REWIND** ステートメントはファイルを初期点に位置付けます。

構文

```
►►─REWIND──┬─u─────────────────┬──────────────────────────────────────►◄
           └─(─position_list─)─┘
```

u
外部装置識別子です。 *u* の値はアスタリスク、またはホレリス定数であってはなりません。

position_list
装置指定子 ([**UNIT=**]*u*) を必ず 1 つ含んでいなければならないリストです。このリストには、他の有効な各指定子を 1 つずつ入れることができます。有効な指定子は次のとおりです。

[UNIT=] *u*
装置指定子です。*u* は外部装置識別子で、その値はアスタリスクであってはなりません。外部装置識別子はスカラー整数式 (1 ～ 2,147,483,647 の値を持つ) で表される外部ファイルを示します。オプションの文字である **UNIT=** を省略する場合、*u* は *position_list* の最初の項目でなければなりません。

IOSTAT= *ios*
I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。 **REWIND** ステートメントの実行が完了すると、*ios* の値は次のように定義されます。
- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値

ERR= *stmt_label*
エラーが発生した場合に制御が移される同じ有効範囲単位内の実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子をコーディングすると、エラー・メッセージは抑制されます。

規則

装置が接続されていない場合、**fort.***n* という名前のデフォルトのファイルに対して順次アクセスを指定する暗黙の **OPEN** ステートメントが実行されます。ここで、*n* は先行ゼロを除去した *u* の値です。指定した装置に接続されている外部ファイルが存在しない場合、**REWIND** ステートメントは効力を持ちません。接続されている外部ファイルが存在している場合、ファイルの終わりマーカーが作成され (必要な場合)、ファイルが最初のレコードの始めに位置付けられます。ファイルがすでに初期点に位置付けられている場合、**REWIND** ステートメントは効力を持ちません。 **REWIND** ステートメントを実行すると、*u* を参照しているそれ以降の **READ** または **WRITE** ステートメントでは、*u* に関連付けられた外部ファイルの最初のレコードからのデータの読み取り、またはそのレコードへのデータの書き込みが行われます。

ERR= と **IOSTAT=** 指定子が設定されているときにエラーが検出されると、**ERR=** 指定子によって指定されたステートメントに対して転送が行われ、正の整数値が *ios* に割り当てられます。

IBM 拡張

IOSTAT= も **ERR=** も指定していない場合は、以下のとおりです。
- 重大なエラーが検出されるとプログラムは停止します。
- 回復可能エラーが検出され、**ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、プログラムは次のステートメントの処理を継続します。オプションが **NO** に設定されていると、プログラムは停止します。

IBM 拡張 の終り

例

```
REWIND (9, IOSTAT=IOSS)
```

関連情報

- 229 ページの『条件および IOSTAT 値』
- 217 ページの『第 8 章 I/O の概念』
- 「ユーザーズ・ガイド」の『*実行時オプションの設定*』

SAVE

目的

SAVE 属性は、変数および名前付き共通ブロックを定義したサブプログラムから制御が戻った後も、定義状況を保持したいオブジェクトおよび名前付き共通ブロックの名前を指定します。

構文

```
>>—SAVE—————————————————————————————————————————————————————————————————><
          |                                       |
          |           ┌—,—————————————————————┐   |
          |           V                       |   |
          └—┬——————┬——┬—object_name———————————┬—┘
            └—::—┘    └—/—common_block_name—/—┘
```

規則

リストを指定していない **SAVE** ステートメントでは、有効範囲単位内のすべての共通項目およびローカル変数の名前をステートメントに指定したかのように処理されます。**SAVE** 属性を持つ共通ブロック名は、その名前付き共通ブロック内のすべてのエンティティーを指定することと同じ効果を持ちます。

関数サブプログラムまたはサブルーチン・サブプログラム内では、**SAVE** 属性で指定した変数は、そのサブプログラムで **RETURN** または **END** ステートメントを実行しても未定義にはなりません。

object_name に仮引き数、ポインティング先、プロシージャー、自動オブジェクト、または共通ブロック・エンティティーの名前を指定することはできません。

サブプログラムで **RETURN** または **END** ステートメントを実行するときに、**SAVE** 属性を持たせて指定したローカル・エンティティー (共通ブロックに入っていないもの) が定義済みの状態であれば、このエンティティーは、同じサブプログラムが次に参照されるときに、同じ値で定義されます。保管されたオブジェクトはサブプログラムのすべてのインスタンスによって共用されます。

IBM 拡張

XL Fortran では、関数結果は **SAVE** 属性を持つことができます。関数結果に **SAVE** 属性を持たせることを示すには、**SAVE** 属性で明示的に関数結果名を指定しなければなりません。つまり、リストを指定していない **SAVE** ステートメントは、関数結果に **SAVE** 属性を提供することはありません。

SAVE として宣言される変数は、スレッド間で共用されます。共用変数を含むアプリケーションをスレッド・セーフにするには、ロックを使用して静的データへのアクセスを逐次化するか、データをスレッド固有にしなければなりません。データをスレッド固有にするための 1 つの方法は、**THREADLOCAL** と宣言された名前付き **COMMON** ブロックに静的データを移動することです。 **Pthreads** ライブラリー・モジュールには、ロックを使ってデータへのアクセスを逐次化するための mutex が備わっています。詳細については、 787 ページの『第 15 章 Pthreads ライブラリー・モジュール』を参照してください。また、**CRITICAL** ディレクティブの *lock_name* 属性にも、データへのアクセスを逐次化するための機能が備わっています。 詳細については、 701 ページの『CRITICAL / END CRITICAL』を参照してください。 **THREADLOCAL** ディレクティブを使うと、確実に共通ブロックをおのおののスレッドに対してローカルにすることができます。詳細については、 738 ページの『THREADLOCAL』を参照してください。

IBM 拡張 の終り

SAVE 属性と互換性のある属性

• ALLOCATABLE	• PRIVATE	• STATIC
• DIMENSION	• PROTECTED	• TARGET
• POINTER	• PUBLIC	• VOLATILE

例

```
LOGICAL :: CALLED=.FALSE.
CALL SUB(CALLED)
CALLED=.TRUE.
CALL SUB(CALLED)
CONTAINS
  SUBROUTINE SUB(CALLED)
    INTEGER, SAVE :: J
    LOGICAL :: CALLED
    IF (CALLED.EQV..FALSE.) THEN
      J=2
    ELSE
      J=J+1
    ENDIF
    PRINT *, J                      ! Output on first call is 2
                                    ! Output on second call is 3
  END SUBROUTINE
END
```

関連情報

- 314 ページの『COMMON』
- 738 ページの『THREADLOCAL』
- 70 ページの『変数の定義状況』
- 78 ページの『変数のストレージ・クラス』
- 923 ページの『付録 A. 異なる標準の間の互換性』の項目 2

SELECT CASE

目的

SELECT CASE ステートメントは **CASE** 構造体の最初のステートメントです。**CASE** 構造では、実行するステートメント・ブロックの中から 1 つのみを選択できるように簡略化された構文を提供しています。

構文

```
►►─┬────────────────────────┬─SELECT CASE─(─case_expr─)─────────────►◄
   └─case_construct_name─:─┘
```

case_construct_name
: **CASE** 構造体を識別する名前です。

case_expr
: 整数タイプ、文字タイプまたは論理タイプのスカラー式です。

規則

SELECT CASE ステートメントを実行すると、*case_expr* が評価されます。結果として得られる値をケース指標といいます。このケース指標は、ケース構造体内の制御の流れを評価するために使用されます。

case_construct_name を指定する場合、構造体内の **END CASE** ステートメントには必ずこの名前を指定しなくてはなりませんが、**CASE** ステートメントへの指定は任意です。

IBM 拡張

case_expr は、タイプなし定数または **BYTE** データ・オブジェクト以外のものでなければなりません。

IBM 拡張 の終り

例

```
ZERO: SELECT CASE(N)         ! start of CASE construct ZERO

  CASE DEFAULT ZERO
       OTHER: SELECT CASE(N) ! start of CASE construct OTHER
          CASE(:-1)
             SIGNUM = -1
          CASE(1:) OTHER
              SIGNUM = 1
       END SELECT OTHER
  CASE (0)
    SIGNUM = 0

END SELECT ZERO
```

関連情報

- 149 ページの『CASE 構造体』
- 304 ページの『CASE』
- **END SELECT** ステートメントの詳細については、 351 ページの『END (構造体)』

SEQUENCE

目的

SEQUENCE ステートメントを指定することによって、派生型定義のコンポーネントの順序でオブジェクトの記憶順序が決定されます。これによって、このタイプは*順序派生型* となります。

構文

```
▶▶──SEQUENCE──────────────────────────────────────────────────────────▶◀
```

規則

SEQUENCE ステートメントは派生型定義の中で 1 回だけ指定することができます。

順序派生型のコンポーネントが派生型の場合、その派生型も順序派生型でなければなりません。

IBM 拡張

順序派生型のサイズは、その派生型のすべてのコンポーネントを保持するために必要なストレージのバイト数に等しくなります。

IBM 拡張 の終り

順序派生型を使用すると、データの並びが揃わなくなることがあります。これは、プログラムのパフォーマンスに悪影響を与えます。

例

```
TYPE PERSON
  SEQUENCE
  CHARACTER*1 GENDER     ! Offset 0
  INTEGER(4) AGE         ! Offset 1
  CHARACTER(30) NAME     ! Offset 5
END TYPE PERSON
```

関連情報

- 41 ページの『派生型』
- 333 ページの『派生型 (TYPE)』
- 356 ページの『END TYPE』

ステートメント関数

目的

ステートメント関数は単一ステートメント内の関数を定義します。

構文

```
>>—name—(————————————————————)— = —scalar_expression——————————————————><
         └dummy_argument_list┘
```

name ステートメント関数の名前です。これをプロシージャー引き数にすることはできません。

dummy_argument
1 つのステートメント関数の仮引き数リストの中で 1 回だけ指定できます。仮引き数には、ステートメント関数ステートメントの有効範囲があります。そのタイプと型付きパラメーターは、ステートメント関数を含む有効範囲単位内で同じ名前を持つエンティティーと同じです。

規則

ステートメント関数は、その関数を定義している有効範囲単位に対してローカルです。ステートメント関数をモジュールの有効範囲内で定義することはできません。

ステートメント関数から戻された値のデータ型は、*name* によって決まります。 *name* のデータ型がスカラー式のデータ型と一致しない場合は、割り当てステートメントの規則に従ってスカラー式の値が *name* のタイプに変換されます。

関数およびすべての仮引き数の名前はスカラー・データ・オブジェクトになるように、明示的にあるいは暗黙的に指定しなければなりません。

スカラー式は定数、変数の参照、関数および関数の仮プロシージャーの参照、および組み込み演算によって構成されます。関数または関数の仮プロシージャーへの参照が式に含まれている場合、参照は明示インターフェースを使用することはできず、関数が明示インターフェースを使用したり変形可能な組み込み演算にすることはできません。結果はスカラーでなければなりません。関数または関数の仮プロシージャーが配列値の場合は、配列名を付ける必要があります。

IBM 拡張

XL Fortran では、スカラー式は構造体コンポーネントを参照することもできます。

IBM 拡張 の終り

スカラー式は、次のステートメント関数のいずれかを参照することができます。
- 同じ有効範囲単位内で前もって宣言された他のステートメント関数
- ホスト有効範囲単位内で宣言された他のステートメント関数

式の中で参照されるエレメントを持つ名前付き定数と配列は、有効範囲単位内で前もって宣言するか、または使用関連付けかホスト関連付けを介してアクセス可能にしなければなりません。

式の中で参照される変数は次のうちのどちらかです。
- ステートメント関数の仮引き数
- 有効範囲単位内でアクセス可能な変数

式の中のエンティティーのタイプが暗黙のタイプ規則で決定されている場合、そのタイプと型付きパラメーターは、それに続くタイプ宣言ステートメント内のものと一致しなければなりません。

スカラー式の中で外部関数を参照することによって、ステートメント関数の仮引き数が未定義あるいは再定義になってはいけません。

ステートメント関数を内部サブプログラム内で定義し、そのステートメント関数とホストからアクセス可能なエンティティーの名前が同じ場合、ステートメント関数を定義する前にステートメント関数の名前を明示的に定義しておく必要があります。たとえば、タイプ宣言ステートメントを使用します。

文字タイプのステートメント関数、または文字タイプのステートメント関数の仮引き数の長さ指定は、定数の宣言式でなければなりません。

例

```
PARAMETER (PI = 3.14159)
REAL AREA,CIRCUM,R,RADIUS
AREA(R) = PI * (R**2)            ! Define statement functions
CIRCUM(R) = 2 * PI * R           !   AREA and CIRCUM

! Reference the statement functions
PRINT *,'The area is: ',AREA(RADIUS)
PRINT *,'The circumference is: ',CIRCUM(RADIUS)
```

関連情報

- 195 ページの『仮引き数』
- 190 ページの『関数参照』
- ステートメント関数のタイプがどのように決まるかについては、 69 ページの『タイプの決め方』

STATIC

IBM 拡張

目的

STATIC 属性を指定すると、変数のストレージ・クラスが静的になります。つまり、プログラムの実行中は変数がメモリー内に残り、その値がプロシージャーの呼び出しの間で保存されます。

構文

```
                    ┌─,──────────────────────────────────────────────┐
                    ▼                                                │
►►──STATIC──┬────┬──stat_variable──┬──────────────────────────────┬─┴──────►◄
            └─::─┘                 └─/──initial_value_list──/─────┘
```

stat_variable
: *explicit_shape_spec_list* または *deferred_shape_spec_list* を指定できる変数名または配列宣言子です。

initial_value
: 直前の名前で指定される変数に初期値を与えます。初期化は、 327 ページの『DATA』の説明のとおりに行われます。

規則

stat_variable が結果変数の場合、文字タイプまたは派生型であってはなりません。仮引き数、自動オブジェクトおよびポインティング先に **STATIC** 属性を指定することはできません。 **STATIC** 属性によって明示的に宣言された変数を共通ブロック項目に指定することはできません。

同じ有効範囲単位内で 1 つの変数に対して **STATIC** 属性を何回も指定することはできません。

ローカル変数はデフォルトで自動ストレージ・クラスを持っています。呼び出しコマンドに関するデフォルト設定値の詳細については、「ユーザーズ・ガイド」の『**-qsave** オプション』を参照してください。

STATIC として宣言される変数は、スレッド間で共用されます。共用変数を含むアプリケーションをスレッド・セーフにするには、ロックを使用して静的データへのアクセスを逐次化するか、データをスレッド固有にしなければなりません。データをスレッド固有にするための 1 つの方法は、**THREADLOCAL** と宣言された **COMMON** ブロックに静的データを移動することです。 **Pthreads** ライブラリー・モジュールには、ロックを使ってデータへのアクセスを逐次化するための mutex が備わっています。詳細について

は、 787 ページの『第 15 章 Pthreads ライブラリー・モジュール』を参照してください。また、**CRITICAL** ディレクティブの *lock_name* 属性にも、データへのアクセスを逐次化するための機能が備わっています。 詳細については、 701 ページの『CRITICAL / END CRITICAL』を参照してください。 **THREADLOCAL** ディレクティブを使うと、確実に共通ブロックをおのおののスレッドに対してローカルにすることができます。詳細については、 738 ページの『THREADLOCAL』を参照してください。

STATIC 属性と互換性のある属性

- ALLOCATABLE
- DIMENSION
- POINTER
- PRIVATE
- PROTECTED
- SAVE
- TARGET
- VOLATILE

例

```
LOGICAL :: CALLED=.FALSE.
CALL SUB(CALLED)
CALLED=.TRUE.
CALL SUB(CALLED)
CONTAINS
  SUBROUTINE SUB(CALLED)
    INTEGER, STATIC :: J
    LOGICAL :: CALLED
    IF (CALLED.EQV..FALSE.) THEN
      J=2
    ELSE
      J=J+1
    ENDIF
    PRINT *, J                  ! Output on first call is 2
                                ! Output on second call is 3
  END SUBROUTINE
END
```

関連情報

- 78 ページの『変数のストレージ・クラス』
- 314 ページの『COMMON』
- 738 ページの『THREADLOCAL』

IBM 拡張 の終り

STOP

目的

STOP ステートメントが実行されると、プログラムは実行を停止します。文字定数または数字ストリングが指定されている場合は、キーワード **STOP** とそれに続けて定数あるいは数字ストリングを装置 0 に出力します。

構文

```
►►──STOP──┬───────────────┬──────────────────────────────────────────►◄
          ├─char_constant─┤
          └─digit_string──┘
```

char_constant
スカラー文字定数です。この値はホレリス定数であってはなりません。

digit_string
1 〜 5 桁からなるストリングです。

規則

IBM 拡張

char_constant も *digit_string* も指定していない場合は、標準エラー (装置 0) には何も出力されません。

IBM 拡張 の終り

STOP ステートメントは、 **DO** または **DO WHILE** の範囲を終了させることはできません。

IBM 拡張

digit_string を指定すると、XL Fortran はシステム戻りコードを **MOD** (*digit_string*,256) に設定します。システム戻りコードを参照するには、コーン・シェルのコマンド変数 *$?* を使用してください。

IBM 拡張 の終り

例

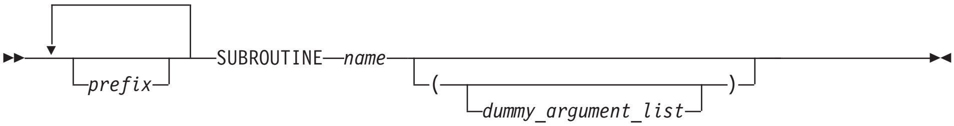
```
STOP 'Abnormal Termination'    ! Output:  STOP Abnormal Termination
END

STOP                           ! No output
END
```

SUBROUTINE

目的

SUBROUTINE ステートメントはサブルーチン・サブプログラムの最初のステートメントです。

構文



prefix 次のうちの 1 つです。

- F95 **ELEMENTAL** F95
- F95 **PURE** F95
- **RECURSIVE**

注: *type_spec* はサブルーチン・サブプログラムの名前です。

name サブルーチン・サブプログラムの名前です。

規則

最大 1 つの *prefix* を指定できます。

反復をあらかじめ指定している場合を除いて、サブルーチン名を、サブルーチンの有効範囲内の他のステートメントに指定することはできません。

以下の場合、キーワード **RECURSIVE** を直接または間接的に指定しなければなりません。
- サブルーチンがそれ自体を呼び出す
- 同じサブプログラムの中の **ENTRY** ステートメントによって定義されたプロシージャーを、サブルーチンが呼び出す
- 同じサブプログラム内の入り口プロシージャーがそれ自体を呼び出す

- 同じサブプログラム内の入り口プロシージャーが同じサブプログラム内の他の入り口プロシージャーを呼び出す
- 同じサブプログラム内の入り口プロシージャーが **SUBROUTINE** ステートメントによって定義されるサブプログラムを呼び出す

キーワード **RECURSIVE** を指定した場合、サブプログラムの中でプロシージャー・インターフェースは明示的になります。

Fortran 95

PURE または**ELEMENTAL** prefix を使用すると、コンパイラーが、順序に関係なく、あたかも副次作用がないかのようにサブルーチンを呼び出すことを示します。 エレメント型プロシージャーでは、キーワード **ELEMENTAL** を指定しなければなりません。 **ELEMENTAL** キーワードを指定している場合、 **RECURSIVE** キーワードを指定することはできません。

Fortran 95 の終り

IBM 拡張

XL Fortran では、**PURE** サブルーチンに以下の 3 つの例外が許されます。
- **OUT** または **INOUT** の意図が指定された仮引き数は修正できる。
- ポインター仮引き数を **INTENT(IN)** として指定していなければ、 **POINTER** 属性を持つ仮引き数の関連付け状況を修正できる。
- **POINTER** 属性を指定した仮引き数の値は修正できる。

-qrecur コンパイラー・オプションを指定すると、外部プロシージャーを再帰的に呼び出すことができます。ただし、**SUBROUTINE** ステートメントがキーワード **RECURSIVE** を指定している場合、XL Fortran はこのオプションを無視します。

IBM 拡張 の終り

例

```
RECURSIVE SUBROUTINE SUB(X,Y)
  INTEGER X,Y
  IF (X.LT.Y) THEN
    RETURN
  ELSE
    CALL SUB(X,Y+1)
  END IF
END SUBROUTINE SUB
```

関連情報

- 188 ページの『関数およびサブルーチン・サブプログラム』
- 195 ページの『仮引き数』

- 209 ページの『再帰』
- 302 ページの『CALL』
- 359 ページの『ENTRY』
- 460 ページの『RETURN』
- 70 ページの『変数の定義状況』
- 209 ページの『純粋プロシージャー』
- 「ユーザーズ・ガイド」の『**-qrecur** オプション』

TARGET

目的

TARGET 属性を持つデータ・オブジェクトはポインターに関連付けることができます。

構文

```
                    ┌─,──────────────────────────────────┐
                    ▼                                    │
►►──TARGET──┬────┬───variable_name──┬──────────────────┬─┴─────────────────────►◄
            └─::─┘                  └─(──array_spec──)─┘
```

規則

データ・オブジェクトが **TARGET** 属性を持つ場合は、すべてのデータ・オブジェクトの非ポインター・サブオブジェクトも **TARGET** 属性を持ちます。

TARGET 属性を持たないデータ・オブジェクトを、アクセス可能なポインターに関連付けることはできません。

ターゲットを **EQUIVALENCE** ステートメント内で指定することはできません。

IBM 拡張

ターゲットを整数ポインターまたはポインティング先にすることはできません。

IBM 拡張 の終り

TARGET 属性と互換性のある属性

• ALLOCATABLE	• OPTIONAL	• SAVE
• AUTOMATIC	• PRIVATE	• STATIC
• DIMENSION	• PROTECTED	• VALUE
• INTENT	• PUBLIC	• VOLATILE

例

```
REAL, POINTER :: A,B
REAL, TARGET  :: C = 3.14
B => C
A => B       ! A points to C
```

関連情報

- 431 ページの『POINTER (Fortran 90)』
- 553 ページの『ALLOCATED(ARRAY) または ALLOCATED(SCALAR)』
- 331 ページの『DEALLOCATE』
- 142 ページの『ポインターの割り当て』
- 167 ページの『ポインター関連付け』

TYPE

目的

タイプ宣言ステートメント **TYPE** は、派生型のオブジェクトと関数のタイプと属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>—TYPE—(—type_name—)——┬—::——————————————————┬—entity_decl_list——————————><
                        └—,—attr_spec_list—::—┘
```

それぞれの意味は次のとおりです。

attr_spec
ALLOCATABLE **AUTOMATIC** **DIMENSION** (*array_spec*) **EXTERNAL** **INTENT** (*intent_spec*) **INTRINSIC** **OPTIONAL** **PARAMETER** **POINTER** **PRIVATE** **PUBLIC** **SAVE** **STATIC** **TARGET** **VOLATILE**

type_name
派生型の名前です。

attr_spec
特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
IN、**OUT**、または **INOUT** のいずれかです。

:: ダブル・コロン・セパレーターです。複数の属性を指定する場合、= *initialization_expr* を使用する場合、 F95 または =>**NULL()** F95 を *entity_decl* の一部として使用する場合に、必要になります。

array_spec
次元境界のリストです。

entity_decl

```
▶▶─a─┬─────────────────┬─┬──────────────────────────────────────┬──────────────────▶◀
     └─(─array_spec─)─┘ │                               (1)     │
                        └─┬─/─initial_value_list─/─────────────┬┘
                          ├─ = ─initialization_expr────────────┤
                          │                      (2)           │
                          └─ => ─NULL()────────────────────────┘
```

注:

1 IBM 拡張

2 Fortran 95

a オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。初期化は、 327 ページの『DATA』の説明のとおりに行われます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。

- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

1 度派生型を定義した後は、タイプ宣言ステートメント **TYPE** で、その派生型を利用してデータ項目を定義することができます。エンティティーを派生型に明示的に宣言する場合、その派生型は前もって有効範囲単位内で定義しておくか、または使用関連付けかホスト関連付けによってアクセス可能になっていなければなりません。

データ・オブジェクトは、*派生型のオブジェクト* か *構造体* になります。おのおのの *構造体コンポーネント* は派生型のオブジェクトのサブオブジェクトです。

DIMENSION 属性を指定すると、データ型が派生型であるエレメントを持つ配列が作成されます。

仕様ステートメント以外では、派生型のオブジェクトを実引き数および仮引き数として使用できます。また、I/O リスト (オブジェクトが、**POINTER** 属性のコンポーネントを持つ場合を除く)、割り当てステートメント、構造体コンストラクター、およびステートメント関数定義の右側の項目として、派生型のオブジェクトを使用することができます。構造体コンポーネントへのアクセスが不可能な場合、派生型のオブジェクトを I/O リストの中で使用することはできません。また、構造体コンストラクターとして使用することもできません。

非順序派生型のオブジェクトを **EQUIVALENCE** および **COMMON** ステートメントのデータ項目として使用することはできません。非順序データ型のオブジェクトは整数のポインティング先にはなれません。

非順序派生型の仮引き数は、使用関連付けまたはホスト関連付けを介してアクセス可能な派生型を指定し、同じ派生型定義が実引き数と仮引き数の両方を定義していることを確認する必要があります。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブ

ジェクトの場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。オブジェクトがブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、 IBM またはモジュール内の名前付き共通ブロックにある場合、そのオブジェクトを初期化することができます。 IBM

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を 1 つのタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、*initialization_expr* F95 または **NULL()** F95 を指定した場合、変数は最初に定義されます。

Fortran 95

宣言するエンティティーが派生型コンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルトの初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。*a* が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、*a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は **DIMENSION** 属性の中の *array_spec* よりも優先します。

POINTER 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。 **FUNCTION** ステートメントで派生型を指定できますが、それは、その派生型が関数の本体の中で定義されているか、あるいは、ホスト関連付けまたは使用関連付けを介してアクセスできる場合に限られます。

IBM 拡張

事前に定数の名前として定義されている T または F がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

例

```
TYPE PEOPLE                      ! Defining derived type PEOPLE
  INTEGER AGE
  CHARACTER*20 NAME
END TYPE PEOPLE
TYPE(PEOPLE) :: SMITH = PEOPLE(25,'John Smith')
END
```

関連情報

- 41 ページの『派生型』
- 333 ページの『派生型 (TYPE)』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』

タイプ宣言

目的

タイプ宣言ステートメントは、オブジェクトおよび関数のタイプ、長さ、属性を指定します。オブジェクトには初期値を割り当てることができます。

構文

```
>>─type_spec─┬─::──────────────────┬─entity_decl_list──────────────><
             └─,─attr_spec_list─::─┘
```

それぞれの意味は次のとおりです。

type_spec	*attr_spec*
BYTE 1	**ALLOCATABLE**
CHARACTER [*char_selector*]	**AUTOMATIC**
COMPLEX [*kind_selector*]	**DIMENSION** (*array_spec*)
DOUBLE COMPLEX 1	**EXTERNAL**
DOUBLE PRECISION	**INTENT** (*intent_spec*)
INTEGER [*kind_selector*]	**INTRINSIC**
LOGICAL [*kind_selector*]	**OPTIONAL**
REAL [*kind_selector*]	**PARAMETER**
TYPE (*type_name*)	**POINTER**
	PRIVATE
	PROTECTED
	PUBLIC
	SAVE
	STATIC
	TARGET
	VALUE
	VOLATILE

注:

1. IBM 拡張

type_name
派生型の名前です。

kind_selector

```
>>─┬─(─┬──────────┬─int_initialization_expr─)─┬────────────────><
   │   └─KIND─ = ─┘                            │
   │                        (1)                │
   └─ * ─int_literal_constant──────────────────┘
```

注:

1 IBM 拡張

関連するタイプに許されている長さ指定のいずれかを示します。

IBM 拡張

int_literal_constant には、kind 型付きパラメーターは指定できません。

IBM 拡張 の終り

char_selector

文字長を指定します。

IBM 拡張

XL Fortranでは、これは 0 から 256 MB までの文字数です。 256 MB を超える値は 256 MB に設定されます。負の値はゼロに設定されます。値を指定しない場合、デフォルトの長さは 1 です。kind 型付きパラメーターを指定した場合、値は 1 でなければなりません。その値は ASCII 文字を表します。

IBM 拡張 の終り

```
>>──┬─(──┬─LEN──=──type_param_value──,──KIND──=──int_init_expr──────────┬─)─┬──────────────><
    │    ├─type_param_value──,──┬────────────┬──int_init_expr────────────┤   │
    │    │                      └─KIND──=────┘                           │   │
    │    ├─KIND──=──int_init_expr──┬──────────────────────────────────┬──┤   │
    │    │                         └─,──LEN──=──type_param_value──────┘  │   │
    │    └──┬─────────┬──type_param_value────────────────────────────────┘   │
    │       └─LEN──=──┘                                                      │
    └─*──char_length──┬───┬──────────────────────────────────────────────────┘
                      └─,─┘
```

type_param_value
: 宣言式またはアスタリスク (*) です。

int_init_expr
: スカラー整数初期化式です。この値は 1 でなければなりません。

char_length
: 括弧で囲んだ、*type_param_value*、またはスカラー整数リテラル定数 (この定数は kind 型付きパラメーターを指定することはできません) です。

attr_spec

特定の属性の規則については、その属性と同じ名前を持つステートメントを参照してください。

intent_spec
: **IN**、**OUT**、または **INOUT** のいずれかです。

::
: ダブル・コロン・セパレーターです。複数の属性を指定する場合、あるいは = *initialization_expr* F95 または => **NULL()** F95 を使用する場合に必要になります。

array_spec
: 次元境界のリストです。

entity_decl

```
>>─a─┬───────────────────────────────────────────┬──────────────────►
     ├─┬──────────────────┬─┬──────────────────┬─┤
     │ └─ * ─char_length─┘ └─(─array_spec─)─┘ │
     └─(─array_spec─)─ * ─char_length──────────┘

►─┬──────────────────────────────────┬──────────────────────────────►◄
  │                             (1)  │
  └─┬─/─initial_value_list─/───────┬─┘
    ├─ = ─initialization_expr──────┤
    │            (2)               │
    └─ => ─NULL()──────────────────┘
```

注:

1　IBM 拡張

2　Fortran 95

a　オブジェクト名または関数名です。 *array_spec* を関数名に指定することはできません。

IBM 拡張

char_length
: *kind_selector* および *char_selector* で指定した長さをオーバーライドします。また、これは先頭のキーワードとして長さが指定されたステートメントの中でだけ認められます。上で定義したように、文字エンティティーは、*char_length* を指定することができます。文字以外のエンティティーは、関連するタイプに認められている長さ指定の 1 つを示す整数リテラル定数のみを指定することができます。

IBM 拡張 の終り

IBM 拡張

initial_value
直前の名前によって指定されるエンティティーに初期値を与えます。

IBM 拡張 の終り

initialization_expr
初期化式を使用して、直前の名前によって指定されるエンティティーに初期値を与えます。

Fortran 95

=> **NULL()**
ポインター・オブジェクトに初期値を与えます。

Fortran 95 の終り

規則

Fortran 95

派生型の定義について、以下のことが該当します。

- => がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定しなければばなりません。
- = がコンポーネントの初期化で現れる場合、 **POINTER** 属性を *attr_spec_list* に指定できません。
- コンパイラーは、タイプ宣言の有効範囲単位内にある *initialization_expr* を評価します。

変数に => を指定する場合、オブジェクトに **POINTER** 属性を指定しなければばなりません。

Fortran 95 の終り

変数に *initialization_expr* を指定する場合、オブジェクトに **POINTER** 属性を指定できません。

タイプ宣言ステートメント中のエンティティーは、エンティティーに対して指定された属性の規則に従います。詳細は対応する属性ステートメントを参照してください。

タイプ宣言ステートメントは、有効な暗黙のタイプの規則をオーバーライドします。組み込み関数のタイプを確認するためにタイプ宣言ステートメントを使用することができ

ます。タイプ宣言ステートメントの中に組み込み関数の総称名または特定名が使用されていても、その名前は組み込み関数の特性を失うことはありません。

オブジェクトが仮引き数、割り振り可能オブジェクト、ポインター、関数結果、無名共通ブロック内のオブジェクト、整数ポインター、外部名、組み込み名、または自動オブジェクトの場合、そのオブジェクトをタイプ宣言ステートメントの中で初期化することはできません。 **AUTOMATIC** 属性を持っているオブジェクトも初期化することはできません。オブジェクトがブロック・データ・プログラム単位内の名前付き共通ブロックにある場合、 IBM またはモジュール内の名前付き共通ブロックにある場合、そのオブジェクトを初期化することができます。 IBM

Fortran 95

Fortran 95 では、ポインターは初期化することができます。ポインターを初期化するには、**=> NULL()** を使用する方法しかありません。

Fortran 95 の終り

インターフェース本体、あるいはサブプログラムの指定部分に宣言式がある場合、*array_spec* または *type_param_value* の宣言式を非定数式にすることができます。この非定数式を使用する宣言オブジェクトのうち、仮引き数またはポインティング先でないものを*自動オブジェクト* といいます。

1 つの属性を特定のタイプ宣言ステートメントの中で繰り返すことはできません。また、1 つの有効範囲単位内でエンティティーに同じ属性を明示的に複数回与えることもできません。

ステートメントが **PARAMETER** 属性を持つときは、*initialization_expr* を指定する必要があります。宣言するエンティティーが変数で、*initialization_expr* F95 または **NULL()** F95 を指定した場合、変数は最初に定義されます。

Fortran 95

宣言するエンティティーが派生型コンポーネントで、 *initialization_expr* または **NULL()** を指定した場合は、派生型にはデフォルト初期化があります。

Fortran 95 の終り

a は、組み込み割り当ての規則に従って、*initialization_expr* によって決まる値により定義されます。エンティティーが配列である場合は、タイプ宣言ステートメントか、または同じ有効範囲単位内の前の仕様ステートメントで、その配列の形状を指定する必要があります。変数または変数サブオブジェクトを複数回初期化することはできません。*a*

が変数で、*initialization_expr* F95 または **NULL()** F95 がある場合、*a* が保管済みオブジェクト (名前付き共通ブロック内のオブジェクトを除く) であることを意味します。オブジェクトの初期化はオブジェクトの基本的なストレージ・クラスに影響を与える場合もあります。

entity_decl の中で指定されている *array_spec* は、**DIMENSION** の中で指定されている *array_spec* に優先します。

ALLOCATABLE または **POINTER** 属性を持たない配列関数の結果は、明示的形状配列の仕様を持つものでなければなりません。

宣言されたエンティティーが関数の場合、アクセス可能な明示インターフェースを持つことはできません。組み込み関数の場合は、この限りではありません。

IBM 拡張

事前に定数の名前として定義されている T または F がタイプ宣言ステートメントにある場合、それは省略された論理定数ではなく名前付き定数の名前になります。

IBM 拡張 の終り

ステートメント内にダブル・コロン・セパレーター (::) がない場合にのみ、**CHARACTER** タイプ宣言ステートメントの *char_length* の後にオプションのコンマを入れることができます。

CHARACTER タイプ宣言ステートメントがモジュール、ブロック・データ・プログラム単位、またはメインプログラムの有効範囲にあり、エンティティーの長さを継承されるものとして指定する場合、そのエンティティーは名前付き文字定数の名前でなければなりません。文字定数は **PARAMETER** 属性に定義される対応する式の長さになります。

CHARACTER タイプ宣言ステートメントがプロシージャーの有効範囲にあり、エンティティーの長さが継承される場合、そのエンティティーの名前は仮引き数または名前付き文字定数の名前でなければなりません。ステートメントが外部関数の有効範囲にある場合、そのステートメントを同じプログラム単位内の **FUNCTION** または **ENTRY** ステートメント内の関数名またはエントリー名にすることができます。エンティティー名が仮引き数の名前の場合、仮引き数はプロシージャーを参照するために、関連する実際の引き数の名前を受け入れます。エンティティー名が文字定数の名前と同じ場合、文字定数は **PARAMETER** 属性が定義する対応した式の長さを受け入れます。エンティティー名が関数名またはエントリー名と同じ場合、エンティティーは呼び出し側の有効範囲単位内で指定されている長さを受け入れます。

文字関数の長さは、関数のタイプがインターフェース・ブロックで宣言されていない場合は定数式でなければならない宣言式になり、仮プロシージャー名の長さを示す場合は

アスタリスクになります。内部関数、モジュール関数、または再帰的関数の場合、または関数が配列またはポインティング値を返す場合、長さにアスタリスクを使うことはできません。

例

```
CHARACTER(KIND=1,LEN=6) APPLES /'APPLES'/
CHARACTER*7, TARGET :: ORANGES = 'ORANGES'
CALL TEST(APPLES)
END

SUBROUTINE  TEST(VARBL)
  CHARACTER*(*), OPTIONAL :: VARBL   ! VARBL inherits a length of 6

  COMPLEX, DIMENSION (2,3) :: ABC(3) ! ABC has 3 (not 6) array elements
  REAL, POINTER :: XCONST

  TYPE PEOPLE                        ! Defining derived type PEOPLE
    INTEGER AGE
    CHARACTER*20 NAME
  END TYPE PEOPLE
  TYPE(PEOPLE) :: SMITH = PEOPLE(25,'John Smith')
END
```

関連情報

- 27 ページの『第 3 章 データ型およびデータ・オブジェクト』
- 110 ページの『初期化式』
- 暗黙の入力規則の詳細については、 69 ページの『タイプの決め方』
- 85 ページの『配列宣言子』
- 29 ページの『自動オブジェクト』
- 78 ページの『変数のストレージ・クラス』
- 初期値の詳細については、 327 ページの『DATA』

USE

目的

USE ステートメントは、モジュールの共通エンティティーへのローカル・アクセスを可能にするモジュール参照です。

構文

```
►►──USE──module_name──┬──────────────────────────┬──────────────────────────►◄
                      ├─,──rename_list───────────┤
                      └─,──ONLY──:──┬───────────┬┘
                                    └─only_list─┘
```

rename アクセス可能なデータ・エンティティーへのローカル名の割り当てです。

```
local_name => use_name
```

only *rename*、総称仕様、または変数、プロシージャー、派生型、名前付き定数、名前リスト・グループの名前です。

規則

USE ステートメントは、*specification_part* の中の他のすべてのステートメントより前の位置にのみ指定できます。 1 つの有効範囲単位内に複数の **USE** ステートメントを指定することができます。

IBM 拡張

USE ステートメントを含むファイルがコンパイルされるとき、指定するモジュールはファイルの中で **USE** ステートメントの前になければなりません。あるいは、そのモジュールが他のファイルにある場合は、前もってコンパイルされていなければなりません。各参照エンティティーはモジュール内の共通エンティティーの名前でなければなりません。

IBM 拡張 の終り

有効範囲単位内のエンティティーはモジュール・エンティティーと*使用関連* の関係になります。また、ローカル・エンティティーは対応するモジュール・エンティティーの属性を持ちます。

PRIVATE 属性の他、**USE** ステートメントの **ONLY** 文節はアクセス可能なモジュールに関する制約を指定します。 **ONLY** 文節を指定すると、*only_list* に指定されているエンティティーだけがアクセス可能になります。キーワードの後にリストを指定しないと、いずれのモジュール・エンティティーもアクセス可能にはなりません。 **ONLY** 文節がない場合、すべての共通エンティティーがアクセス可能になります。

1 つの有効範囲単位内に同一のモジュールを指定する **USE** ステートメントが複数存在し、そのうちの 1 つのステートメントに **ONLY** 文節がない場合、すべての共通エンティティーがアクセス可能になります。 **USE** ステートメントのそれぞれに **ONLY** 文節がある場合、1 つ以上の *only_lists* に指定されているエンティティーだけがアクセス可能になります。

ローカルな使用を目的としてアクセス可能なエンティティーの名前を変更することができます。モジュール・エンティティーは複数のローカル名でアクセスすることができます。名前変更を指定しない場合、使用関連のエンティティーの名前はローカル名になります。使用関連のローカル名を再度宣言することはできません。しかし、**USE** ステートメントかモジュールの有効範囲単位内に存在する場合は、ローカル名を **PUBLIC** または **PRIVATE** ステートメントに指定することができます。

ある有効範囲単位にアクセス可能な複数の総称インターフェースの名前、演算子、および割り当てが同じ場合、それらは 1 つの総称インターフェースとして処理されます。そのような場合、総称インターフェースのうちの 1 つに、同じ名前を持つアクセス可能なプロシージャーへのインターフェース本体を含ませることができます。それ以外の場合は、名前が有効範囲単位内のエンティティーを参照するために使用されていないときにだけ、任意の 2 つの使用関連のエンティティーに同じ名前を指定することができます。使用関連のエンティティーとホスト・エンティティーが同じ名前を共用する場合、ホスト・エンティティーはその名前を使用したホスト関連付けを介してもアクセスできないようになります。

モジュールは、直接的にも間接的にもそれ自身を参照することはできません。たとえば、モジュール Y がモジュール X を参照する場合、モジュール X はモジュール Y を参照することはできません。

モジュール (たとえばモジュール B) が使用関連付けを介して他のモジュール (たとえばモジュール A) の共通エンティティーにアクセスすることを考えてください。モジュール B のローカル・エンティティー (モジュール A からのエンティティーと使用関連の関係を持つエンティティーを含みます) の他のプログラム単位へのアクセス可能度は **PRIVATE** および **PUBLIC** 属性、また、それらを指定していない場合は、モジュール B のデフォルトのアクセス可能度で決定されます。もちろん、他のプログラム単位はモジュール A の共通エンティティーに直接アクセスすることができます。

例

```
MODULE A
  REAL :: X=5.0
END MODULE A
MODULE B
  USE A
  PRIVATE :: X                 !  X cannot be accessed through module B
  REAL :: C=80, D=50
END MODULE B
PROGRAM TEST
  INTEGER :: TX=7
  CALL SUB
  CONTAINS

  SUBROUTINE SUB
  USE B, ONLY : C
  USE B, T1 => C
  USE B, TX => C               !  C is given another local name
  USE A
  PRINT *, TX                  !  Value written is 80 because use-associated
                               !  entity overrides host entity
  END SUBROUTINE
END
```

関連情報

- 183 ページの『モジュール』
- 438 ページの『PRIVATE』
- 444 ページの『PUBLIC』
- 24 ページの『ステートメントおよび実行の順序』

VALUE

IBM 拡張

目的

VALUE 属性は、仮引き数と実引き数の間の引き数関連付けを指定します。この関連付けによって、仮引き数に実引き数の値を渡すことができます。 Fortran 2000 ドラフト標準で、値による引き渡しインプリメンテーションにおいては、**%VAL** 組み込み関数の標準準拠オプションが提供されています。

実引き数、および関連付けられた仮引き数は、単独で変更できます。仮引き数の値または定義状況に対する変更は実引き数に影響を与えません。 **VALUE** 属性を持つ仮引き数は、実引き数の値と等しい初期値を持つ一時変数と関連付けられます。

構文

```
>>──VALUE──┬──────┬──dummy_argument_name_list──────────────────────><
           └─ :: ─┘
```

規則

VALUE 属性は仮引き数にのみ指定する必要があります。

%VAL または **%REF** 組み込み関数を使用して、**VALUE** 属性を持つ仮引き数または関連付けられた実引き数を参照してはなりません。

VALUE 属性を持つ仮引き数を指定したプロシージャーの参照では、明示的インターフェースが必要です。

length パラメーターを省略した場合、または 1 の値を持つ初期化式を使用して指定した場合、 **VALUE** 属性を持つ仮引き数は文字タイプになります。

VALUE 属性を以下に指定してはなりません。

- 配列
- **ALLOCATABLE** コンポーネントを持つ派生型
- 仮プロシージャー

VALUE 属性と互換性のある属性

- INTENT(IN)
- OPTIONAL
- TARGET

仮引き数が **VALUE** 属性と **TARGET** 属性の両方を持つ場合、仮引き数に関連付けられたポインターはプロシージャーの実行後に未定義となります。

例

```
Program validexm1
  integer :: x = 10, y = 20
  print *, 'before calling: ', x, y
  call intersub(x, y)
  print *, 'after calling: ', x, y

  contains
  subroutine intersub(x,y)
    integer, value ::  x
    integer y
    x = x + y
    y = x*y
    print *, 'in subroutine after changing: ', x, y
  end subroutine
end program validexm1
```

次のような出力になります。

```
before calling: 10 20
in subroutine after changing: 30 600
after calling: 10 600
```

関連情報

%VAL 組み込み関数の詳細については、198 ページを参照してください。

IBM 拡張 の終り

VIRTUAL

IBM 拡張

目的

VIRTUAL ステートメントは配列の名前と次元を指定します。 **VIRTUAL** ステートメントは **DIMENSION** ステートメントの代替形式ですが、**VIRTUAL** 属性はありません。

構文

```
►►—VIRTUAL—array_declarator_list——————————————————————————————————►◄
```

規則

配列の次元は最大 20 個まで指定することができます。

1 つの有効範囲単位内では 1 つの配列名に 1 度だけ配列指定をすることができます。

例

```
VIRTUAL A(10), ARRAY(5,5,5), LIST(10,100)
VIRTUAL ARRAY2(1:5,1:5,1:5), LIST2(I,M)       ! adjustable array
VIRTUAL B(0:24), C(-4:2), DATA(0:9,-5:4,10)
VIRTUAL ARRAY (M*N*J,*)                        ! assumed-size array
```

関連情報

- 83 ページの『第 4 章 配列の概念』
- 334 ページの『DIMENSION』

IBM 拡張 の終り

VOLATILE

IBM 拡張

目的

VOLATILE 属性を指定することにより、データ・オブジェクトを独立した I/O プロセスおよび独立した非同期割り込みプロセスがアクセスできるメモリーにマップできます。揮発性データ・オブジェクトを操作するコードは最適化されません。

構文

```
                          ┌─,──────────────────────┐
                          ▼                        │
►►──VOLATILE──┬──────┬────┬─variable_name──────────┴──────────────────────►◄
              └─::───┘    ├─/──common_block_name──/─┤
                          └─derived_type_name───────┘
```

規則

配列名を揮発性 (volatile) として宣言すると、その配列の各エレメントも揮発性と見なされます。共通ブロックを揮発性として宣言すると、その共通ブロックの各変数も揮発性と見なされます。共通ブロック内のその他のエレメントの状況に影響を与えずに、共通ブロックの 1 つのエレメントだけを揮発性として宣言することもできます。

1 つの共通ブロックを複数の有効範囲内で宣言する場合、および 1 つの共通ブロック (または共通ブロックの 1 つ以上のエレメント) を複数の有効範囲のうちの 1 つで揮発性として宣言する場合、その共通ブロック (または共通ブロックの 1 つ以上のエレメント) を揮発性と見なすそれぞれの有効範囲内に **VOLATILE** 属性を指定しなければなりません。

派生型名を揮発性として宣言すると、そのタイプで宣言されたすべての変数は揮発性と見なされます。派生型のオブジェクトを揮発性として宣言すると、そのコンポーネントもすべて揮発性と見なされます。派生型のコンポーネント自体が派生型である場合は、そのコンポーネントはそのタイプから揮発属性を継承しません。揮発性として宣言されている派生型名には、タイプ宣言ステートメント内のタイプ名を使用する前に **VOLATILE** 属性を指定しなければなりません。

ポインターを揮発性として宣言すると、そのポインターのストレージ自体が揮発性と見なされます。 **VOLATILE** 属性は、関連するポインター・ターゲットには影響を与えません。

あるオブジェクトを揮発性として宣言し、そのオブジェクトを **EQUIVALENCE** ステートメントで使用した場合、等価関連付けによってその揮発性オブジェクトに関連したオブジェクトも、すべて揮発性と見なされます。

スレッドを介して共用され、複数のスレッドによって保管され、読み取られるデータ・オブジェクトは、**VOLATILE** として宣言されなければなりません。ただし、プログラムがコンパイラーの自動またはディレクティブ・ベースの並列機能のみを使用する場合、**SHARED** 属性を持つ変数は、**VOLATILE** と宣言する必要はありません。

仮引き数に関連した実引き数が揮発性と宣言されている変数の場合、仮引き数も揮発性と見なす場合は、仮引き数を揮発性として宣言しなければなりません。仮引き数を揮発性と宣言しているときに、関連する実引き数も揮発性と見なす場合は、実引き数を揮発性として宣言しなければなりません。

ステートメント関数を揮発性として宣言しても、そのステートメント関数には影響を与えません。

関数サブプログラム内では、関数結果変数を揮発性として宣言することができます。すべての入力結果変数は揮発性と見なされます。 **ENTRY** 名を **VOLATILE** 属性を使って指定することはできません。

VOLATILE 属性と互換性のある属性

- ALLOCATABLE
- AUTOMATIC
- DIMENSION
- INTENT
- OPTIONAL
- POINTER
- PRIVATE
- PROTECTED
- PUBLIC
- SAVE
- STATIC
- TARGET

例

```
      FUNCTION TEST ()
        REAL ONE, TWO, THREE
        COMMON /BLOCK1/A, B, C
        ...
        VOLATILE /BLOCK1/, ONE, TEST
! Common block elements A, B and C are considered volatile
! since common block BLOCK1 is declared volatile.
        ...
        EQUIVALENCE (ONE, TWO), (TWO, THREE)
! Variables TWO and THREE are volatile as they are equivalenced
! with variable ONE which is declared volatile.
      END FUNCTION
```

関連情報

507 ページの『第 11 章 汎用ディレクティブ』

IBM 拡張 の終り

WAIT

IBM 拡張

目的

WAIT ステートメントは、非同期データ転送の完了を待つため、または非同期データ転送ステートメントの完了状況を検出するために使用することができます。

構文

```
►►—WAIT—(—wait_list—)————————————————————————————————————————————————►◄
```

wait_list
: 1 つの **ID=** 指定子と、他の有効な指定子をそれぞれ最大で 1 つ入れなければならないリストです。有効な指定子は次のとおりです。

ID= *integer_expr*
: **WAIT** ステートメントで識別されるデータ転送を示します。 *integer_expr* は、**INTEGER(4)** またはデフォルト整数のタイプのスカラーです。非同期データ転送を開始するために、**ID=** 指定子は、**READ** または **WRITE** ステートメント上で使用されます。

IOSTAT= *ios*
: I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。この指定子を含む I/O ステートメントの実行が完了すると、*ios* は以下の値に定義されます。

- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値
- ファイルの終わりが検出されていて、しかも、エラーが発生しなかった場合は、負の値が定義されます。

非同期データ転送ステートメントの **IOSTAT=** 指定子に対して定義された *ios* は、対応する **WAIT** ステートメントの **IOSTAT=** に対して定義された *ios* と同一である必要はありません。

ERR= *stmt_label*
: エラーが発生した場合に制御が移される、同じ有効範囲単位内の実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子をコーディングすると、エラー・メッセージは抑制されます。

非同期データ転送ステートメントの **ERR=** 指定子に対して定義された *stmt_label* は、対応する **WAIT** ステートメントの **ERR=** 指定子に対して定義された *stmt_label* と同一である必要はありません。

END= *stmt_label*
エラーが発生せずにファイル終了レコードまで達した場合に、プログラムの実行を継続するステートメント・ラベルを指定するファイルの終わり指定子です。外部ファイルがファイルの最終レコードの後に位置付けられると、**IOSTAT=** 指定子が指定されている場合、この指定子には負の値が割り当てられます。 **NUM=** 指定子が指定されている場合、この指定子には整数値が割り当てられます。 **END=** 指定子をコーディングすると、ファイルの終わりに関するエラー・メッセージが抑止されます。この指定子は、順次アクセスまたは直接アクセス用に接続された装置に指定することができます。

非同期データ転送ステートメントの **END=** 指定子に対して定義された *stmt_label* は、対応する **WAIT** ステートメントの **END=** 指定子に対して定義された *stmt_label* と同一である必要はありません。

DONE= *logical_variable*
非同期 I/O ステートメントが完了するかしないかを指定します。 **DONE=** 指定子がある場合、*logical_variable* は、非同期 I/O が完了する場合には true、完了しない場合には false に設定されます。戻り値が false の場合、**DONE=** 指定子がないか、その戻り値が true になるまで、1 つまたは複数の **WAIT** ステートメントを実行する必要があります。 **DONE=** 指定子を持たない **WAIT** ステートメント、または *logical_variable* 値を ture に設定する **WAIT** ステートメントは、同じ **ID=** 値によって識別されるデータ転送ステートメントへの **WAIT** ステートメントです。

規則

対応する **WAIT** ステートメントは、対応する非同期データ転送ステートメントと同じ有効範囲単位内になければなりません。有効範囲単位内のインスタンス内では、プログラムは、対応する **WAIT** ステートメントが実行される前に、**RETURN**、**END**、または **STOP** ステートメントを実行することはできません。

関連情報

225 ページの『データ転送ステートメントの非同期の実行』
「ユーザーズ・ガイド」の『*XL Fortran I/O のインプリメンテーションの詳細*』

IBM 拡張 の終り

WHERE

目的

WHERE ステートメントは、配列式および配列割り当ての計算をマスクします。これは論理配列式の値に応じて実行されます。 **WHERE** ステートメントは **WHERE** 構造体の最初のステートメントにすることができます。

構文

```
                                            (1)
>>-+-------------------------+-----------------WHERE--(--mask_expr--)--+------------------------------+-------><
   '-where_construct_name--:-'                                          '-where_assignment_statement-'
```

注:

1 Fortran 95 *(where_construct_name)*。

mask_expr
: 論理配列式です。

Fortran 95

where_construct_name
: **WHERE** 構造体を識別する名前です。

Fortran 95 の終り

規則

where_assignment_statement がある場合は、**WHERE** ステートメントは、**WHERE** 構造体の最初のステートメントではありません。 *where_assignment_statement* がない場合は、**WHERE** ステートメントは **WHERE** 構造体の最初のステートメントであり、**WHERE** 構造体ステートメントと呼ばれます。 **END WHERE** ステートメントをそれに続けなければなりません。 詳細については、 131 ページの『WHERE 構造体』を参照してください。

WHERE ステートメントが **WHERE** 構造体の最初のステートメントではない場合、そのステートメントを **DO** または **DO WHILE** 構造体の終端ステートメントとして使用することができます。

Fortran 95

WHERE ステートメントは、**WHERE** 構造体内でネストさせることができます。定義済み割り当てである *where_assignment_statement* は、エレメント型の定義済み割り当てでなければなりません。

Fortran 95 の終り

where_assignment_statement では、 *mask_expr* および定義されている変数 *variable* は、同じ形状の配列でなければなりません。**WHERE** 構造体内のそれぞれの *mask_expr* は同じ形状でなければなりません。

Fortran 95

where_body_construct の一部である **WHERE** ステートメントは、分岐のターゲット・ステートメントにすることはできません。

Fortran 95 の終り

WHERE ステートメントの *mask_expr* で参照される関数の実行は、*where_assignment_statement* 内のエンティティーに影響を与える場合があります。

マスク式の解釈については、 133 ページの『マスクされた配列割り当ての解釈』 を参照してください。

Fortran 95

where_construct_name を **WHERE** 構造体ステートメントに指定する場合、対応する **END WHERE** ステートメントにも指定しなければなりません。構造体名は、**WHERE** 構造体のマスクされた **ELSEWHERE** および **ELSEWHERE** ステートメントでは任意指定です。

where_construct_name は、**WHERE** 構造体ステートメント上だけに指定可能です。

Fortran 95 の終り

例

```
REAL, DIMENSION(10) :: A,B,C

!   In the following WHERE statement, the LOG of an element of A
!   is assigned to the corresponding element of B only if that
!   element of A is a positive value.

WHERE (A>0.0) B = LOG(A)

   ⋮
END
```

Fortran 95

以下の例は、**WHERE** ステートメントでのエレメント型の定義済み割り当てを示しています。

```
INTERFACE ASSIGNMENT(=)
  ELEMENTAL SUBROUTINE MY_ASSIGNMENT(X, Y)
    LOGICAL, INTENT(OUT) :: X
    REAL, INTENT(IN) :: Y
  END SUBROUTINE MY_ASSIGNMENT
END INTERFACE
```

```
INTEGER A(10)
REAL C(10)
LOGICAL L_ARR(10)

C = (/ -10., 15.2, 25.5, -37.8, 274.8, 1.1, -37.8, -36.2, 140.1, 127.4 /)
A = (/ 1, 2, 7, 8, 3, 4, 9, 10, 5, 6 /)
L_ARR = .FALSE.

WHERE (A < 5) L_ARR = C

! DATA IN ARRAY L_ARR AT THIS POINT:
!
! L_ARR = F, T, F, F, T, T, F, F, F, F

END

ELEMENTAL SUBROUTINE MY_ASSIGNMENT(X, Y)
  LOGICAL, INTENT(OUT) :: X
  REAL, INTENT(IN) :: Y

  IF (Y < 0.0) THEN
    X = .FALSE.
  ELSE
    X = .TRUE.
  ENDIF
END SUBROUTINE MY_ASSIGNMENT
```

Fortran 95 の終り

関連情報

- 131 ページの『WHERE 構造体』
- 348 ページの『ELSEWHERE』
- **END WHERE** ステートメントの詳細については、 351 ページの『END (構造体)』

WRITE

目的

WRITE ステートメントはデータ転送出力ステートメントです。

構文

```
►►──WRITE──(──io_control_list──)──┬──────────────────┬──────────────►◄
                                  └─output_item_list─┘
```

output_item

出力リスト項目です。出力リストには転送するデータを指定します。出力リスト項目には、次のものを指定できます。

- 変数名。この配列は、すべての配列エレメントが、ストレージに並んでいる順序で指定されているかのように処理されます。

  ポインターはターゲットと関連付ける必要があり、割り振り可能オブジェクトは割り振る必要があります。派生型のオブジェクトは、このステートメントの有効範囲単位の外側にある最終コンポーネントを持つことはできません。 *output_item* を評価しても、ポインターを含む派生型のオブジェクトは発生しません。定様式ステートメント内の構造体のコンポーネントは、派生型定義で現れる順序で指定されているかのように処理されます。不定様式ステートメントでは、構造体コンポーネントは内部表示の 1 つの値として処理されます (埋め込みを含みます)。
- 式
- 暗黙 **DO** リスト。詳細は 504 ページの『暗黙 DO リスト』に記載されています。

io_control

装置指定子 (**UNIT=**) を必ず 1 つ含んでいなければならないリストです。このリストには、他の有効な指定子をそれぞれ 1 つずつ入れることができます。

[UNIT=] *u*

出力操作で使用する装置を指定する装置識別子です。 *u* は、外部装置識別子または内部ファイル識別子です。

IBM 拡張

外部装置識別子は外部ファイルを示します。それは次のうちの 1 つです。

- 値が 0 〜 2,147,483,647 の範囲内にある整数式。
- アスタリスク。外部装置 6 を識別し、標準出力にあらかじめ接続されているもの。

IBM 拡張 の終り

内部ファイル識別子は内部ファイルを示します。これは、ベクトル添え字を持つ配列セクションにはならない文字変数の名前です。

オプションの文字である **UNIT=** を省略する場合、*u* は *io_control_list* の最初の項目でなければなりません。**UNIT=** を指定する場合、**FMT=** も指定する必要があります。

[FMT=] *format*

出力操作で使用する形式を指定する形式指定子です。 *format* は形式識別子で、次のいずれかです。

- **FORMAT** ステートメントのステートメント・ラベル。 **FORMAT** ステートメントは同じ有効範囲単位内になければなりません。
- スカラー **INTEGER(4)** または **INTEGER(8)** 変数の名前。これには **FORMAT** ステートメントのステートメント・ラベルが割り当てられています。 **FORMAT** ステートメントは同じ有効範囲単位内になければなりません。

Fortran 95

Fortran 95 ではステートメント・ラベルの割り当ては行えません。

Fortran 95 の終り

- 括弧の付いた文字定数。両括弧の間で使用できるのは、 372 ページの『FORMAT』に記述した形式コードだけです。ブランク文字は左括弧の前または右括弧の後ろにあってもかまいません。
- 左端の文字位置の部分が有効な形式になっている文字データを含む文字変数。有効な形式とは左括弧で始まり、右括弧で終わる形式です。両括弧の間で使用できるのは、**FORMAT** ステートメントに記述した形式コードだけです。ブランク文字は左括弧の前または右括弧の後ろにあってもかまいません。 *format* が配列エレメントの場合、形式識別子の長さは配列エレメントの長さを超えてはなりません。
- 非文字組み込みタイプの配列。文字配列の項で説明したように、データは有効な形式識別子でなければなりません。
- 文字式。ただし、オペランドが定数の名前でない場合、長さの継承を指定するオペランドの連結を含む文字式を除きます。
- リスト指示形式設定を指定するアスタリスク。
- 事前に定義した名前リストのリスト名を指定する名前リスト指定子。

オプションの文字である **FMT=** を省略する場合、*io_control_list* 内の 2 番目の項目は *format* でなければなりません。最初の項目は、**UNIT=** を省略した装置指定子でなければなりません。 1 つの出力ステートメントに **NML=** と **FMT=** を両方とも指定することはできません。

POS=*integer_expr*

integer_expr はスカラー整数式で、式の値は 0 より大きくなければなりません。**POS=** はストリーム・アクセス用に接続されたファイル内で書き込まれるファイル記憶単位のファイル位置を示します。 **POS=** は、位置決めを行うことができないファイルに使用してはなりません。

REC= *integer_expr*

直接アクセス用に接続されたファイル内で書き込みを実行したいレコードの番号を指定するレコード指定子です。 **REC=** 指定子を使用できるのは、直接出力の場合に限られます。 *integer_expr* は正の値を持つ整数式です。形式設定が

リスト指示の場合、または装置指定子で内部ファイルを指定している場合、レコード指定子は有効ではありません。レコード指定子は、ファイル内のレコードの相対的な位置を示します。最初のレコードの相対位置番号は 1 です。ストリーム・アクセス用に接続された装置を指定するデータ転送ステートメントで **REC=** を指定してはなりません。また、**POS=** 指定子を使用してはなりません。

IOSTAT= *ios*

I/O 操作の状況を示す I/O 状況指定子です。 *ios* は、**INTEGER(4)** タイプのスカラー変数またはデフォルトの整数です。 **IOSTAT=** 指定子は、エラー・メッセージを抑制します。ステートメントの実行が完了すると、*ios* の値は次のように定義されます。

- エラーが発生しなかった場合は 0
- エラーが発生した場合は正の値

IBM 拡張

ID= *integer_variable*

データ転送が非同期に行われることを示します。 *integer_variable* は、**INTEGER(4)** またはデフォルト整数のタイプのスカラーです。エラーが検出されない場合、*integer_variable* は、非同期データ転送ステートメントの実行後に 1 つの値で定義されます。この値は、対応する **WAIT** ステートメント内で使用されなければなりません。

非同期データ転送は、直接不定様式、順次不定様式、ストリーム不定様式のいずれかでなければなりません。内部ファイルへの非同期 I/O は禁止されています。ロー文字装置への非同期 I/O (たとえば、テープまたはロー論理ボリュームへの非同期 I/O) は、禁止されています。 *integer_variable* を、データ転送 I/O リストのエンティティーや、データ転送 I/O リストの *io_implied_do* の *do_variable* に関連させることはできません。 *integer_variable* が配列エレメント参照の場合、その添え字値は、データ転送、*io_implied_do* 処理、または *io_control_spec* 内の他の指定子の定義や評価などによって影響を受けてはなりません。

IBM 拡張 の終り

ERR= *stmt_label*

エラーが発生した場合に制御が移される同じ有効範囲単位内の実行可能ステートメントのステートメント・ラベルを指定するエラー指定子です。 **ERR=** 指定子をコーディングすると、エラー・メッセージは抑制されます。

IBM 拡張

NUM= *integer_variable*
入出力リストとファイルの間で転送されるデータのバイト数を指定する数指定子です。 *integer_variable* は、**INTEGER(4)** タイプ、64 ビットの **INTEGER(8)** タイプ、またはデフォルトの整数タイプの変数名です。 **NUM=** 指定子を使用できるのは、不定様式出力の場合に限られます。 **NUM** パラメーターをコーディングすると、出力リストに表示されるバイト数が、レコードに書き込めるバイト数よりも大きい場合、出されるエラー表示は抑止されます。この場合、*integer_variable* の値は、書き込み可能な最大レコード長に設定されます。残りの出力リスト項目からのデータは、これ以後のレコードには書き込まれません。非同期データ転送ステートメントと、対応する **WAIT** ステートメントとの間で非同期データ転送ステートメントを実行するプログラムの一部では、**NUM=** 指定子の *integer_variable* またはそれに関連するすべての変数を、参照したり、定義したり、定義を削除したりしてはなりません。

IBM 拡張 の終り

[NML=] *name*
事前に定義した名前リストの名前を指定する名前リスト指定子です。オプションの文字である **NML=** を指定しない場合、リストの 2 番目のパラメーターとして名前リストの名前でなければなりません。また、最初の項目は **UNIT=** を省略した装置指定子でなければなりません。 **NML=** と **UNIT=** の両方を指定する場合、すべてのパラメーターを任意の順序で指定することができます。 **NML=** 指定子は **FMT=** の代替指定子です。 1 つの出力ステートメントに **NML=** と **FMT=** の両方とも指定することはできません。

ADVANCE= *char_expr*
このステートメントについて非事前出力が発生するかどうかを決定する事前指定子です。 *char_expr* は、文字式で、式の値は **YES** または **NO** のいずれかになります。**NO** を指定した場合、非事前出力が発生します。 **YES** を指定した場合、事前定様式順次出力または事前定様式ストリーム出力が発生します。デフォルト値は **YES** です。内部ファイル単位指定子を指定しない明示的な形式仕様を持つ定様式の順次 **WRITE** ステートメントにだけ、**ADVANCE=** を指定できます。

暗黙 DO リスト

```
►►─(─do_object_list─ , ─do_variable = arith_expr1, arith_expr2──────────►

►─┬───┬─┬─────────────┬─)────────────────────────────────────────────►◄
  └─,─┘ └─arith_expr3─┘
```

do_object
出力リスト項目です。

do_variable
　整数、または実数タイプのスカラー変数です。

arith_expr1、*arith_expr2*、および *arith_expr3*
　スカラー数式です。

暗黙 **DO** リストの範囲は *do_object_list* です。繰り返し回数および **DO** 変数の値は、**DO** ステートメントの場合と同様に、*arith_expr1*、 *arith_expr2*、および *arith_expr3* で決まります。暗黙 **DO** リストが実行されると、暗黙 **DO** リストの繰り返しごとに、*do_object_list* 内の項目が 1 つ指定され、**DO** 変数のその時点の値に応じた適切な値に置き換えられます。

規則

IBM 拡張

NUM= 指定子を指定した場合は、形式指定子も名前リスト指定子も指定することはできません。

IBM 拡張 の終り

IOSTAT= および **NUM=** 指定子に指定された変数を、出力リスト項目、名前リストのリスト項目、および暗黙 **DO** リストの **DO** 変数のいずれにも関連付けることはできません。このような指定子変数が配列エレメントの場合、データ転送、暗黙 **DO** 処理、または他の指定子の定義または評価が、その添え字値に影響を与えてはいけません。

ERR= と **IOSTAT=** 指定子が設定されているときにエラーが検出されると、**ERR=** 指定子によって指定されたステートメントに対して転送が行われ、正の整数値が *ios* に割り当てられます。

IBM 拡張

ERR= か **IOSTAT=** 指定子が設定され、非同期データ転送中にエラーが検出されると、対応する **WAIT** ステートメントの実行は要求されません。

変換エラーが検出され、 **CNVERR** 実行時オプションが **NO** に設定されている場合、**IOSTAT=** は設定されますが、 **ERR=** には分岐しません。

IOSTAT= も **ERR=** も指定していない場合は、以下のとおりです。
- 重大なエラーが検出されるとプログラムは停止します。
- 回復可能エラーが検出され、 **ERR_RECOVERY** 実行時オプションが **YES** に設定されていると、プログラムは次のステートメントへと処理を継続します。オプションが **NO** に設定されていると、プログラムは停止します。
- **ERR_RECOVERY** 実行時オプションが **YES** に設定されている場合、変換エラーが検出されると、プログラムは次のステートメントの処理を継続します。 **CNVERR** 実

行時オプションが **YES** に設定されている場合、変換エラーは回復可能エラーとして処理されます。一方、**CNVERR=NO** の場合、エラーは変換エラーとして処理されます。

IBM 拡張 の終り

`PRINT format` は `WRITE(*,format)` と同じ働きをします。

例

```
WRITE (6,FMT='(10F8.2)') (LOG(A(I)),I=1,N+9,K),G
```

関連情報

- 225 ページの『データ転送ステートメントの非同期の実行』
- 「ユーザーズ・ガイド」の『*XL Fortran I/O のインプリメンテーションの詳細*』
- 229 ページの『条件および IOSTAT 値』
- 217 ページの『第 8 章 I/O の概念』
- 445 ページの『READ』
- 496 ページの『WAIT』
- 「ユーザーズ・ガイド」の『*I/O 用の実行時オプションの設定*』
- 927 ページの『削除された機能』

第 11 章 汎用ディレクティブ

IBM 拡張

この章では、すべてのプラットフォームに適用される非 SMP ディレクティブをアルファベット順に説明します。SMP ディレクティブとスレッド・セーフ・ディレクティブの完全なリストと説明については、 695 ページの『第 13 章 SMP ディレクティブ』 を参照してください。また、PowerPC プラットフォーム限定のディレクティブの詳しい説明については、 905 ページの『ハードウェア固有のディレクティブ』を参照してください。この章には以下の節があります。

- 『注釈形式および非注釈形式ディレクティブ』
- 510 ページの『ディレクティブおよび最適化』
- 511 ページの『ディレクティブの詳細説明』

注釈形式および非注釈形式ディレクティブ

XL Fortran ディレクティブは、注釈形式ディレクティブまたは非注釈形式ディレクティブのどちらかのグループに属します。

注釈形式ディレクティブ

この章では、以下の注釈形式ディレクティブについて詳しく説明します。

COLLAPSE	**SNAPSHOT**
SOURCEFORM	**SUBSCRIPTORDER**

この章で説明するもの以外の注釈形式ディレクティブについては、 510 ページの『ディレクティブおよび最適化』を参照してください。

形式

```
►►—trigger_head—trigger_constant—directive——————————————————————►◄
```

trigger_head
: 固定ソース形式の場合は **!**、*****、**C**、または **c** のいずれか 1 つであり、自由ソース形式の場合は **!** です。

trigger_constant
: デフォルトで **IBM*** です。 **-qsmp** コンパイラー・オプションが指定されている場合、 **IBM***、**IBMT**、**SMP$**、**$OMP**、および **IBMP** はデ

フォルトで認識されます。**-qdirective** コンパイラー・オプションを指定すると、他のトリガー定数を定義できるようになります。

規則

trigger_constant のデフォルト値は **IBM*** です。

-qsmp コンパイラー・オプションをこれらの呼び出しコマンドと併用する場合、オプション **-qdirective=IBM*:SMP$:$OMP:IBMP:IBMT** がデフォルトでオンになります。**-qsmp=omp** オプションを指定した場合、デフォルトでオプション **-qdirective=$OMP** を設定したときと同じようになります。**-qdirective** コンパイラー・オプションで代替または追加の *trigger_constant* を指定することもできます。詳細については、「ユーザーズ・ガイド」の **-qdirective** コンパイラー・オプションを参照してください。

すべての注釈形式ディレクティブ (デフォルト *trigger_constant* を使用するディレクティブ以外) は、コンパイラーによって注釈として見なされます。ただし、該当する *trigger_constant* が **-qdirective** コンパイラー・オプションによって定義されている場合は例外です。結果として、これらのディレクティブを含むコードは、非 SMP 環境に移植することができます。

XL Fortran は、IBM での解釈による OpenMP 使用をサポートします。コードの移植性を最大限にするために、可能な限りこれらのディレクティブを使用することをお勧めします。これらのディレクティブは、OpenMP の *trigger_constant*、 **$OMP** と一緒に使用してください。この *trigger_constant* は、他のディレクティブとは使用しないでください。

XLF にはさらに *trigger_constant* の **IBMP** および **IBMT** が含まれます。**-qsmp** コンパイラー・オプションを使用する場合、コンパイラーは **IBMP** を認識します。 **IBMP** は、**SCHEDULE** ディレクティブ、および OpenMP ディレクティブの IBM 拡張と一緒に使用してください。 **-qthreaded** コンパイラー・オプションを使用してコンパイルする場合、コンパイラーは **IBMT** を認識します。 **xlf_r**、 **xlf_r7**、 **xlf90_r**、 **xlf90_r7**、 **xlf95_r** 、または **xlf95_r7** 呼び出しコマンドのデフォルトは **IBMT** です。これは **THREADLOCAL** ディレクティブと一緒に使用することをお勧めします。

XLF ディレクティブには、他のベンダーと共通のディレクティブがあります。これらのディレクティブをコードで使用すると、ベンダーが選択したどの *trigger_constant* でも使用することができます。 **-qdirective** コンパイラー・オプションを使用してトリガー定数を指定すると、ベンダーが選択した *trigger_constant* を使用可能にすることができます。代替 *trigger_constant* の指定について詳しくは、「ユーザーズ・ガイド」の **-qdirective** コンパイラー・オプションを参照してください。

trigger_head は、Fortran 90 の自由ソース形式または固定ソース形式のいずれかの注釈行の規則に従います。 *trigger_head* が **!** である場合、桁 1 に入れる必要はありません。 *trigger_head* と *trigger_constant* の間にブランクを入れないでください。

directive_trigger (*trigger_head* を *trigger_constant*、**!IBM*** と組み合わせて定義したものなど)、およびすべてのディレクティブ・キーワードは、大文字だけ、小文字だけ、大文字小文字の混合のいずれでも指定できます。

インライン注釈をディレクティブ行に指定できます。

```
!SMP$ INDEPENDENT, NEW(i)    !This is a comment
```

ディレクティブは、同じ行の別のステートメントまたは行の後に入れることはできません。

すべての注釈形式ディレクティブは、継続行を持てます。ディレクティブを後続のステートメントに組み込んだり、ステートメントを後続のディレクティブに組み込むことはできません。

すべての継続行において *directive_trigger* を指定しなければなりません。ただし、継続行上の *directive_trigger* は、継続行で使用されている *directive_trigger* と同一である必要はありません。たとえば、次のようにします。

```
!SMP$ INDEPENDENT &
!IBM*&  , REDUCTION (X)                        &
!SMP$&  , NEW (I)
```

これは、 **IBM*** と **SMP$** の両方がアクティブ *trigger_constant* である場合、以下のものと同等です。

```
!SMP$ INDEPENDENT, REDUCTION (X), NEW (I)
```

詳細については、 14 ページの『行およびソース形式』を参照してください。

ディレクティブは、自由ソース形式または固定ソース形式の注釈として (現行のソース形式によって異なる) 指定できます。

固定ソース形式の規則: *trigger_head* が **C**、**c**、***** のいずれかである場合は、1 桁目に入れる必要があります。

1 行または複数行に継続ディレクティブの場合、固定ソース形式の *trigger_constant* の最大長は、4 です。この規則は継続行にのみ適用され、最初の行には適用されません。最初の行の場合、*trigger_constant* の最大長は 15 です。ただし、最初の行のトリガーの最大長は 4 にすることをお勧めします。許容最大長の 15 は後方互換性のために設定されているものです。

trigger_constant の長さが 4 以下の場合、注釈ディレクティブの最初の行の 6 桁目には空白文字またはゼロのどちらかが必要です。長さが 5 以上の場合、6 桁目にある文字は *trigger_constant* の一部になります。

注釈ディレクティブの継続行の *directive_trigger* は、 1 〜 5 桁目に入れる必要があります。継続行の 6 桁目には、空白文字やゼロ以外の文字が必要です。

詳細については、 16 ページの『固定ソース形式』を参照してください。

自由ソース形式の規則: *trigger_constant* の最大長は 15 です。

行の最後のアンパーサンド (&) は、ディレクティブが続くことを示します。 ディレクティブ行を続ける場合は、*directive_trigger* をすべての継続行の最初に入れなければなりません。継続行をアンパーサンドで始める場合でも、*directive_trigger* をアンパーサンドの前に入れる必要があります。たとえば、次のようにします。

```
!IBM* INDEPENDENT &
!SMP$&  , REDUCTION (X)          &
!IBM*&  , NEW (I)
```

詳細については、 19 ページの『自由ソース形式』を参照してください。

非注釈形式ディレクティブ

この章では、以下の非注釈形式ディレクティブについて詳しく説明します。

EJECT	**INCLUDE**
#LINE	**@PROCESS**

形式

```
►►──directive──────────────────────────────────────────────►◄
```

規則

コンパイラーは、非注釈形式のディレクティブを必ず認識します。

非注釈形式ディレクティブは継続行を持つことができません。

付加的なステートメントをディレクティブと同じ行に入れることはできません。

空白文字に関するソース形式規則は、ディレクティブ行に適用されます。

ディレクティブおよび最適化

以下に示すのは、ソース・コードの最適化に便利な注釈形式ディレクティブです。 XL Fortran プログラムの最適化、およびパフォーマンスに影響のあるコンパイラー・オプションの情報については、「ユーザーズ・ガイド」を参照してください。

断定ディレクティブ

断定ディレクティブは、コンパイラーが入手できない、ソース・コードに関する情報を収集します。この情報を提供することにより、パフォーマンスを向上することができます。

ASSERT	**CNCALL**
INDEPENDENT	**PERMUTATION**

ループ・アンロール用ディレクティブ

以下のディレクティブは、ソース・コード内の DO CONSTRUCT の効果を最適化するための、さまざまな方法のループ・アンロールを提供します。

STREAM_UNROLL	**UNROLL**
UNROLL_AND_FUSE	

ディレクティブの詳細説明

ASSERT

ASSERT ディレクティブはコンパイラーに、**DO** ループの特性に関する情報を提供します。これは、コンパイラーがソース・コードを最適化するのに役立ちます。
ASSERT ディレクティブが有効なのは、**-qsmp** または **-qhot** コンパイラー・オプションのどちらかが指定されている場合に限られます。

構文

```
►►—ASSERT—(—assertion—)——————————————————————————————————————►◄
```

assertion

ITERCNT(*n*) または **NODEPS** です。**ITERCNT**(*n*) と **NODEPS** は、互いに排他的ではなく、同じ **DO** ループに両方を指定できます。同じ **DO** ループでは、各引き数を最大 1 つまで指定できます。

ITERCNT(*n*)

n には、指定した **DO** ループの繰り返し回数を指定します。 *n* は、正のスカラー整数初期化式でなければなりません。

NODEPS

指定した **DO** 内にループによる依存性が存在しないことを指定します。

規則

ASSERT ディレクティブに続く最初の非注釈行 (他のディレクティブは含まない) は、**DO** ループでなければなりません。この行は、無限の **DO** または **DO WHILE** ループにすることはできません。 **ASSERT** ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ネストされた **DO** ループには適用されません。

ITERCNT は、**DO** ループが通常実行する繰り返し回数のおおまかな見積もりをコンパイラーに提供します。この値は正確である必要はありません。 **ITERCNT** はパフォーマンスにのみ影響し、正確度には影響しません。

NODEPS を指定すると、ユーザーは、**DO** ループ内、または **DO** ループ内から呼び出されたプロシージャー内に、ループによる依存性が存在しないことを明示的にコンパイラーに宣言したことになります。ループによる依存性には、**DO** ループ内にあって相互に妨害する 2 つの繰り返しが関係しています。妨害は、次の状況で生じます。

- 同じアトミック・オブジェクト (サブオブジェクトがないデータ) を定義、定義解除、または再定義する 2 つの操作は互いに妨害し合います。
- アトミック・オブジェクトを定義、未定義、または再定義は、そのオブジェクトの値の使用と互いに妨害し合います。
- ポインターの関連付け状況を定義または定義解除する操作を行うと、ポインターへの参照や、関連付け状況を定義または定義解除する別の操作は妨害されます。
- **DO** ループの外に制御を移したり、 **EXIT**、**STOP**、**PAUSE** ステートメントを実行すると、他のすべての繰り返しは妨害されます。
- 同じファイルまたは外部装置に関連する 2 つの I/O 操作が存在した場合、これらは相互に妨害し合います。ただし、この場合は以下のような例外があります。
  - 2 つの I/O 操作が 2 つの **INQUIRE** ステートメントである場合。
  - 2 つの I/O 操作が 1 つのストリーム・アクセス・ファイルの別々の領域にアクセスする場合。
  - 2 つの I/O 操作が直接アクセス・ファイルの別々のレコードにアクセスする場合。
- 繰り返し間で割り振り可能オブジェクトの割り振り状況を変更すると、妨害が生じます。

互いに補完する 2 つの **ASSERT** ディレクティブを 1 つの **DO** ループに適用することは可能です。しかし、**ASSERT** ディレクティブの後に、同一の **DO** ループの矛盾した **ASSERT** ディレクティブを入れることはできません。

```
!SMP$ ASSERT (ITERCNT(10))
!SMP$ INDEPENDENT, REDUCTION (A)
!SMP$ ASSERT (ITERCNT(20))     ! invalid
      DO I = 1, N
         A(I) = A(I) * I
      END DO
```

上記の例では、**ASSERT(ITERCNT(20))** ディレクティブは、**ASSERT(ITERCNT(10))** ディレクティブと矛盾しているので、無効です。

ASSERT ディレクティブは、**ASSERT** ディレクティブが指定されている **DO** ループの **-qassert** コンパイラー・オプションをオーバーライドします。

例

例 1:

```
! An example of the ASSERT directive with NODEPS.
         PROGRAM EX1
           INTEGER A(100)
  !SMP$    ASSERT (NODEPS)
           DO I = 1, 100
             A(I) = A(I) * FNC1(I)
           END DO
         END PROGRAM EX1

         FUNCTION FNC1(I)
           FNC1 = I * I
         END FUNCTION FNC1
```

例 2:

```
! An example of the ASSERT directive with NODEPS and
ITERCNT.
         SUBROUTINE SUB2 (N)
           INTEGER A(N)
    !SMP$  ASSERT (NODEPS,ITERCNT(100))
           DO I = 1, N
             A(I) = A(I) * FNC2(I)
           END DO
         END SUBROUTINE SUB2

         FUNCTION FNC2 (I)
           FNC2 = I * I
         END FUNCTION FNC2
```

関連情報

- 「ユーザーズ・ガイド」の『**-qassert** オプション』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qhot** オプション』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 335 ページの『DO』
- 369 ページの『ループの並列化』

CNCALL

CNCALL ディレクティブを **DO** ループの前に置くと、ユーザーは、**DO** ループから呼び出されたプロシージャー内にループによる依存性が存在しないことをコンパイラーに

明示的に宣言したことになります。 **CNCALL** ディレクティブが有効なのは、**-qsmp** または **-qhot** コンパイラー・オプションのいずれかが指定されているときに限られます。

構文

```
►►──CNCALL──────────────────────────────────────────────────────────►◄
```

規則

CNCALL ディレクティブに続く最初の非注釈行は、 **DO** ループでなければなりません。この行は、無限の **DO** または **DO WHILE** ループにすることはできません。 **CNCALL** ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ネストされた **DO** ループには適用されません。

CNCALL ディレクティブを指定すると、ユーザーは、**DO** ループ内で呼び出されたプロシージャーにはループによる依存性がないことをコンパイラーに明示的に宣言したことになります。 **DO** ループがプロシージャーを呼び出す場合、ループのそれぞれの繰り返しがそのプロシージャーを同時に呼び出せなければなりません。 **CNCALL** ディレクティブは、ループ内の他の操作に依存性がないと断定するものではありません。これは単にプロシージャーの参照についての断定です。

ループによる依存性は、**DO** ループ内にある 2 つの繰り返しが互いに妨害するときに生じます。妨害の定義については、 511 ページの『ASSERT』を参照してください。

例

```
! An example of CNCALL where the procedure invoked has
! no loop-carried dependency but the code within the
! DO loop itself has a loop-carried dependency.
         PROGRAM EX3
           INTEGER A(100)
    !SMP$  CNCALL
           DO I = 1, N
             A(I) = A(I) * FNC3(I)
             A(I) = A(I) + A(I-1)    ! This has loop-carried dependency
           END DO
         END PROGRAM EX3

         FUNCTION FNC3 (I)
           FNC3 = I * I
         END FUNCTION FNC3
```

関連情報

- 519 ページの『INDEPENDENT』
- 「ユーザーズ・ガイド」の『**-qdirective**』

- 「ユーザーズ・ガイド」の『**-qhot** オプション』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 335 ページの『DO』
- 369 ページの『ループの並列化』

COLLAPSE

COLLAPSE ディレクティブは、配列次元の下限のエレメントのみがアクセス可能であると指定することによって、配列次元全体を単一のエレメントまでに削減します。下限を指定しない場合は、デフォルトの 下限は 1 になります。

慎重に使用すれば、**COLLAPSE** ディレクティブは、複数次元の配列に関連した反復メモリー・アクセスを削減することによって、容易にパフォーマンスを向上させることができます。

構文

```
▸▸──COLLAPSE──(──collapse_array_list──)──────────────────────────────▸◂
```

collapse_array の意味は次のとおりです。

```
▸▸──array_name──(──expression_list──)────────────────────────────────▸◂
```

ここで *expression_list* は、コンマで区切られた式のリストです。

array name
: 配列名です。

expression
: 定数スカラー整数式です。正整数値のみを指定できます。

規則

COLLAPSE ディレクティブには、最低でも 1 つの配列を含む必要があります。

COLLAPSE ディレクティブは、そのディレクティブが指定されている有効範囲単位に対してのみ適用されます。 **COLLAPSE** ディレクティブに含まれる配列の宣言は、そのディレクティブと同じ有効範囲単位内にある必要があります。使用関連付けまたはホスト関連付けによって有効範囲単位内でアクセス可能な配列は、その有効範囲単位内の **COLLAPSE** ディレクティブに指定してはなりません。

expression_list に指定できる下限の値は 1 です。上限の値は、対応する配列内の次元数より大きくすることはできません。

単一の有効範囲単位に複数の **COLLAPSE** 宣言を含むことができますが、配列は特定の有効範囲単位について一度だけしか指定できません。

1 つの配列を **COLLAPSE** ディレクティブと **EQUIVALENCE** ステートメントの両方に指定することはできません。

COLLAPSE ディレクティブは、派生型のコンポーネントである配列と一緒に使用することはできません。

1 つの配列に対して **COLLAPSE** と **SUBSCRIPTORDER** の両方のディレクティブを適用する場合は、最初に **SUBSCRIPTORDER** ディレクティブを指定する必要があります。

COLLAPSE ディレクティブは、以下の配列に適用されます。

- すべての下限が定数式でなければならない想定形状配列。
- すべての下限が定数式でなければならない明示的形状配列。

例

例 1: 以下の例では、**COLLAPSE** ディレクティブは明示的形状配列 *A* および *B* に適用されます。内部ループの *A(m,2:100,2:100)* と *B(m,2:100,2:100)* の参照は、 *A(m,1,1)* と *B(m,1,1)* になります。

```
!IBM* COLLAPSE(A(2,3),B(2,3))
      REAL*8 A(5,100,100), B(5,100,100), c(5,100,100)

      DO I=1,100
       DO J=1,100
        DO M=1,5
           A(M,J,I) = SIN(C(M,J,I))
           B(M,J,I) = COS(C(M,J,I))
        END DO
        DO M=1,5
         DO N=1,M
            C(M,J,I) = C(M,J,I) + A(N,J,I)*B(6-N,J,I)
         END DO
        END DO
       END DO
      END DO
      END
```

関連情報

SUBSCRIPTORDER ディレクティブの詳細については、 532 ページの『SUBSCRIPTORDER』を参照してください

EJECT

EJECT は、ソース・リストの新しいページを開始するようコンパイラーに指示するためのものです。ソース・リストの要求がない場合、コンパイラーは、このディレクティブを無視します。

構文

```
►►──EJECT──────────────────────────────────────────────────────────►◄
```

規則

EJECT コンパイラー・ディレクティブには、インライン注釈やラベルを組み込めます。しかし、ステートメント・ラベルを指定しても、コンパイラーはそれを破棄します。したがって、**EJECT** ディレクティブ内のラベルを参照してはなりません。 **EJECT** ディレクティブの使用例としては、リスト内の複数ページに分割したくない重要な **DO** ループがある場合、そのループの前で使用します。ソース・リストをプリンターに送信すると、**EJECT** ディレクティブは改ページの役目を果たします。

INCLUDE

INCLUDE コンパイラー・ディレクティブは、指定されたステートメントまたは一群のステートメントをプログラム単位に挿入します。

構文

```
►►──INCLUDE──┬─char_literal_constant─┬──┬───┬──────────────────────►◄
             └─(──name──)────────────┘  └─n─┘
```

name、*char_literal_constant* (区切り文字はオプショナル)
: *filename*、つまりインクルード・ファイルの名前を指定します。

  AIX オペレーティング・システムの場合、ファイルの絶対パスを指定する必要はありません。ただし、ファイル拡張子がある場合は、拡張子を指定する必要があります。

  name には、XL Fortran の文字セットで使用可能な文字のみを入れます。 XL Fortran でサポートしている文字については、 11 ページの『文字』を参照してください。

  char_literal_constant は文字リテラル定数です。

n
: コンパイル時にファイルを組み込むかどうかを決めるために、コンパイラーが使用する値です。 1 から 255 の範囲内であればどの数でもかまいませんが、kind 型付きパラメーターを指定することはできません。 *n* を指定すると、その数が **-qci** (条件付き組み込み) コンパイラー・オプションでサブオプションとして指定されている場合に限り、コンパイラーはファイルを組み込みます。 *n* を指定しなければ、コンパイラーは常にファイルを組み込みます。

条件付き **INCLUDE** と呼ばれる機能は、コンパイル時に、Fortran ソース内の **INCLUDE** コンパイラー・ディレクティブを選択的に活動化状態にする方法を提供します。インクルード・ファイルの指定は、コンパイラー・オプション **-qci** を使用して行います。

固定ソース形式では、**INCLUDE** コンパイラー・ディレクティブは列 6 から始める必要があります。ラベルも付けられます。

インライン注釈を **INCLUDE** 行に追加することができます。

規則

インクルード・ファイルには、完全な Fortran ソース・ステートメントまたはコンパイラー・ディレクティブであれば、どのようなステートメント (他の **INCLUDE** コンパイラー・ディレクティブも含む) でも入れることができます。再帰的 **INCLUDE** コンパイラー・ディレクティブは使用できません。 **END** ステートメントは組み込まれるグループに含めてもかまいません。最初および最後の組み込み行は、継続行であってはなりません。インクルード・ファイルのステートメントは、組み込み先のファイルのソース形式で処理されます。

SOURCEFORM ディレクティブがインクルード・ファイル内にある場合、インクルード・ファイルの処理が完了すると、ソース形式は、組み込み先のファイルのソース形式に戻ります。すべてのグループを組み込んで完成した Fortran プログラムは、ステートメントの順序に関する Fortran のすべての規則に従うものでなければなりません。

XL Fortran は、左括弧と右括弧の構文を持つ **INCLUDE** コンパイラー・ディレクティブについて、**-qmixed** コンパイラー・オプションがオンになっている場合を除いて、ファイル名を小文字に変換します。

AIX ファイル・システムは、*filename* で指定されたファイルを次のようにして見つけます。

- *filename* の最初の非ブランク文字が **/** である場合、 *filename* は絶対ファイル名になります。
- 最初の非ブランク文字が **/** でない場合、 AIX オペレーティング・システムは優先順位の高い方から降順でディレクトリーを探索します。
  - **-I** コンパイラー・オプションを指定すると、指定したディレクトリー内で *filename* が探索されます。
  - AIX オペレーティング・システムが *filename* を見つけることができない場合は、以下を検索します。
    - 現行ディレクトリーで *filename* というファイルを探索します。
    - コンパイルを行うソース・ファイルの常駐ディレクトリーで *filename* というファイルを探索します。
    - `/xlf/8.1/include` ディレクトリーで *filename* というファイルを探索します。

例

```
INCLUDE '/u/userid/dc101'     ! full absolute file name specified
INCLUDE '/u/userid/dc102.inc' ! INCLUDE file name has an extension
INCLUDE 'userid/dc103'        ! relative path name specified

INCLUDE (ABCdef)              ! includes file abcdef

INCLUDE '../Abc'              ! includes file Abc from parent directory
                              ! of directory being searched
```

関連情報

「ユーザーズ・ガイド」の『**-qci** オプション』

INDEPENDENT

INDEPENDENT ディレクティブを使用する場合は、 **DO** ループ、**FORALL** ステートメント、または **FORALL** 構造体の前に置かなければなりません。 **INDEPENDENT** ディレクティブは、**FORALL** ステートメントまたは **FORALL** 構造体の各演算をどの順序で行ってもプログラムのセマンティクスに影響しないで済むようにします。このディレクティブは、プログラムのセマンティクスに影響することなく、 **DO** ループ内で任意の順序で反復を行うことを指定します。
INDEPENDENT ディレクティブが有効なのは、**-qsmp** または **-qhot** コンパイラー・オプションのいずれかが指定されているときに限られます。

構文

```
                 ┌──────────────────────────────────────────┐
                 ▼                                          │
►►──INDEPENDENT──┬─────────────────────────────────────────┬┴──────────────►◄
                 ├─,──NEW──(──named_variable_list──)───────┤
                 └─,──REDUCTION──(──named_variable_list──)─┘
```

規則

INDEPENDENT に続く最初の非注釈行 (他のディレクティブは含まない) は、**DO** ループ、 **FORALL** ステートメント、または **FORALL** 構造体の最初のステートメントでなければなりません。この行は、無限の **DO** または **DO WHILE** ループにすることはできません。 **INDEPENDENT** ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ネストされた **DO** ループには適用されません。

INDEPENDENT ディレクティブには、最大で 1 つの **NEW** 文節と最大で 1 つの **REDUCTION** 文節を組み込めます。

このディレクティブを **DO** ループに適用すると、ループの繰り返しは互いに妨害することはありません。妨害は、次の状況で生じます。

- 2 つのオペレーションが同じアトミック・オブジェクト (サブオブジェクトがないデータ) を定義、定義解除、または再定義すると、妨害が生じます。ただし、**NEW** 文節または **REDUCTION** 文節に親オブジェクトを示している場合は例外です。ネストされた **DO** ループの指標変数を **NEW** 文節に定義する必要があります。
- アトミック・オブジェクトを定義、未定義、または再定義すると、オブジェクトの値の使用が妨害されます。例外は、親オブジェクトが **NEW** 文節または **REDUCTION** 文節にある場合です。
- ポインターの関連付け状況を定義または定義解除する操作を行うと、ポインターへの参照や、関連付け状況を定義または定義解除する別の操作が妨害されます。
- **DO** ループの外に制御を移したり、 **EXIT**、**STOP**、**PAUSE** ステートメントを実行すると、他のすべての繰り返しが妨害されます。
- 同じファイルまたは外部装置に関連する 2 つの I/O 操作が存在した場合、これらは相互に妨害します。ただし、この場合は以下のような例外があります。
  - 2 つの I/O 操作が 2 つの **INQUIRE** ステートメントである場合。
  - 2 つの I/O 操作が 1 つのストリーム・アクセス・ファイルの別々の領域にアクセスする場合。
  - 2 つの I/O 操作が直接アクセス・ファイルの別々のレコードにアクセスする場合。
- 繰り返し間で割り振り可能オブジェクトの割り振り状況を変更すると、妨害が生じます。

NEW 文節を指定する場合は、このディレクティブを **DO** ループに適用する必要があります。 **NEW** 文節は、このディレクティブとそのまわりにある **INDEPENDENT** ディレクティブを修正します。この修正は、**NEW** 文節がそのようなディレクティブによる断定を正しいと見なすことによって生じるもので、 **NEW** 文節で指定されている変数がループの繰り返しごとに修正される場合でも発生します。 **NEW** 文節で指定されている変数は、**DO** ループ本体のプライベート変数であるかのような働きをします。つまり、これらの変数 (およびそれと関連している変数) がループの繰り返しの前後に未定義になっても、プログラムには影響がありません。

NEW 文節または **REDUCTION** 文節で指定する変数には、次の制約事項があります。
- 仮引き数であってはならない
- ポインティング先であってはならない
- 使用関連付けまたはホスト関連付けであってはならない
- 共通ブロック変数であってはならない
- **SAVE** または **STATIC** 属性を指定してはならない
- **POINTER** または **TARGET** 属性を指定してはならない
- **EQUIVALENCE** ステートメントに入れてはならない

FORALL の場合、**INDEPENDENT** ディレクティブによって影響を受けた指標値の組み合わせは、他の組み合わせによって要求されるまではアトミック記憶単位に割り振りを

行いません。 **DO** ループ、**FORALL** ステートメント、または **FORALL** 構造体が同じ本体を持ち、それぞれの前に **INDEPENDENT** ディレクティブがある場合、これらはみな同じ機能を果たします。

REDUCTION 文節は、**INDEPENDENT** ループの **REDUCTION** ステートメント内で名前付き変数が更新されたと断定します。さらに並列セクション内では、**REDUCTION** 変数の中間値は、更新そのもの以外では使用されません。したがって、構造体の後の **REDUCTION** 変数の値は、縮約ツリーの結果になります。

REDUCTION 文節を指定する場合は、このディレクティブを **DO** ループに適用しなければなりません。 **INDEPENDENT DO** ループの **REDUCTION** 変数に対する唯一の参照は、縮約ステートメントになければなりません。

REDUCTION 変数は、組み込みタイプでなければなりませんが、タイプ文字であってはなりません。 **REDUCTION** 変数は割り振り可能配列にはできません。

REDUCTION 変数は、以下のものの中に入れることはできません。
- 同一の **INDEPENDENT** ディレクティブにある **NEW** 文節
- 後続の **DO** ループの本体にある **INDEPENDENT** ディレクティブの **NEW** または **REDUCTION** 文節
- 後続の **DO** ループの本体にある **PARALLEL DO** ディレクティブの **FIRSTPRIVATE**、**PRIVATE**、または **LASTPRIVATE** 文節
- 後続の **DO** ループの本体にある **PARALLEL DO** ディレクティブの **PRIVATE** 文節

REDUCTION ステートメントの形式は、次のいずれかになります。

```
>>--reduction_var_ref--=--expr--reduction_op--reduction_var_ref-------------><

>>--reduction_var_ref--=--reduction_var_ref--reduction_op--expr-------------><

>>--reduction_var_ref =--reduction_function--(expr,--reduction_var_ref)-----><

>>--reduction_var_ref =--reduction_function--(reduction_var_ref,--expr)-----><
```

それぞれの意味は次のとおりです。

reduction_var_ref
: **REDUCTION** 文節に入れる変数または変数のサブオブジェクトです。

reduction_op
: **+**、**-**、*****、**.AND.**、**.OR.**、 **.EQV.**、**.NEQV.**、または **.XOR.** のいずれかです。

reduction_function
: **MAX**、 **MIN**、 **IAND**、 **IOR**、または **IEOR** のいずれかです。

REDUCTION ステートメントには次の規則が適用されます。

1. 縮約ステートメントは、**INDEPENDENT DO** ループの範囲内に入れる割り当てステートメントです。 **REDUCTION** 文節の変数は、**INDEPENDENT DO** ループ内の **REDUCTION** ステートメントにのみ入れることができます。
2. **REDUCTION** ステートメントに入れる 2 つの *reduction_var_ref* は、字句的に同一でなければなりません。
3. **INDEPENDENT** ディレクティブの構文では、配列エレメントまたは配列セクションを **REDUCTION** 文節の **REDUCTION** 変数として指定することはできません。そのようなサブオブジェクトが **REDUCTION STATEMENT** ステートメントに入っている場合でも **REDUCTION** 変数として扱われるのは配列全体です。
4. 次の形式の **REDUCTION** ステートメントは使えません。

```
►►—reduction_var_ref— = —expr— - —reduction_var_ref——————————————————►◄
```

例

例 1:

```
      INTEGER A(10),B(10,12),F
!IBM* INDEPENDENT                       ! The NEW clause cannot be
      FORALL (I=1:9:2) A(I)=A(I+1)      ! specified before a FORALL
!IBM* INDEPENDENT, NEW(J)
      DO M=1,10
        J=F(M)                          ! 'J' is used as a scratch
        A(M)=J*J                        ! variable in the loop
!IBM*   INDEPENDENT, NEW(N)
        DO N=1,12                       ! The first executable statement
          B(M,N)=M+N*N                  ! following the INDEPENDENT must
        END DO                          ! be either a DO or FORALL
      END DO
      END
```

例 2:

```
      X=0
!IBM* INDEPENDENT, REDUCTION(X)
      DO J = 1, M
        X = X + J**2
      END DO
```

例 3:

```
      INTEGER A(100), B(100, 100)
!SMP$ INDEPENDENT, REDUCTION(A), NEW(J)   ! Example showing an array used
      DO I=1,100                          ! for a reduction variable
        DO J=1, 100
          A(I)=A(I)+B(J, I)
        END DO
      END DO
```

関連情報

- 369 ページの『ループの並列化』
- 152 ページの『DO 構造体』
- 367 ページの『FORALL』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qhot** オプション』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

#LINE

#line ディレクティブは、cpp によって作成されるコードまたは Fortran ソース・コード生成プログラムを、プログラマーによって作成される入力コードと関連付けます。プリプロセッサーによりコード行が挿入されたり削除されたりすることがありますが、**#line** ディレクティブはオリジナルのソースのどの行からプリプロセッサーが中間ファイルの対応する行を生成したかを示すので、エラーの報告やデバッグの際に便利です。

構文

```
>>--#line--line_number--+----------+-----------------------------------><
                        '-filename-'
```

#line ディレクティブは非注釈ディレクティブであり、このタイプのディレクティブの構文規則に従います。

line_number
: 正の符号なし整数リテラル定数です。 **KIND** パラメーターは指定できません。*line_number* を指定しなければなりません。

filename
: 文字リテラル定数です。 kind 型付きパラメーターは指定できません。*filename* は、絶対パスまたは相対パスを指定します。指定された *filename* は、後で使用するために保管されます。相対パスを指定すると、プログラムのデバッグ時にデバッガーでディレクトリー検索リストが使用されて *filename* が解決されます。

規則

#line ディレクティブは、他の非注釈ディレクティブと同じ規則に従います。ただし、次のものは例外です。

- **#line** ディレクティブと同じ行にインライン注釈を入れることはできません。
- 自由ソース形式では、**#** 文字と **line** 間のホワイト・スペースはオプションです。
- 固定または自由ソース形式では、語 **line** の文字間に、ホワイト・スペースが組み込まれないことがあります。

- 固定ソース形式では、行のどこからでも **#line** ディレクティブを開始できます。

#line ディレクティブは、現行ファイルのそのディレクティブに続くすべてのコードの起点を示します。別の **#line** ディレクティブが現れると、前のものは取り消されます。

filename を指定すると、現行ファイル内の続くコードは、その起点がその *filename* からであるときと同じようになります。 *filename* を省略し、現行ファイル内のそれよりも前に *filename* を指定した **#line** ディレクティブが存在しない場合、現行ファイルのコードは、指定された行番号の現行ファイルが起点になっているかのように扱われます。*filename* が指定されている、前の **#line** ディレクティブが現行ファイルに存在する場合、その前のディレクティブの *filename* が使用されます。

line_number は、適切なファイル内の、ディレクティブに続くコードの行の位置を示します。そのファイル内の続く行は、別の **#line** ディレクティブが指定されるか、またはファイルが終了するまで、ソース・ファイルにある続く行と 1 対 1 で対応していると想定されます。

XL Fortran がファイルに対して cpp を起動するときに、プリプロセッサーは、**#line** ディレクティブを出力します。これは、**-d** オプションを指定すると行われません。

例

ファイル test.F の内容は以下のとおりです。

```
! File test.F, Line 1
#include "test.h"
PRINT*, "test.F Line 3"
...
PRINT*, "test.F Line 6"
#include "test.h"
PRINT*, "test.F Line 8"
END
```

ファイル test.h の内容は以下のとおりです。

```
!  File test.h line 1
RRINT*,1           ! Syntax Error
PRINT*,2
```

C プリプロセッサー (**/lib/cpp**) で、デフォルト・オプションを指定してファイル test.F を処理した後は次のようになります。

```
#line 1 "test.F"
! File test.F, Line 1
#line 1 "test.h"
!  File test.h Line 1
RRINT*,1           ! Syntax Error
PRINT*,2
#line 3 "test.F"
PRINT*, "test.F Line 3"
...
#line 6
```

```
PRINT*, "test.F Line 6"
#line 1 "test.h"
!  File test.h Line 1
RRINT*,1           ! Syntax Error
PRINT*,2
#line 8 "test.F"
PRINT*, "test.F Line 8"
END
```

コンパイラーは、C プリプロセッサーによって作成されたファイルを処理した後に、次のメッセージを表示します。

```
2       2 |RRINT*,1
!Syntax error
            ......a................
a - "t.h", line 2.6: 1515-019 (S) Syntax is incorrect.

4       2 |RRINT*,1           !Syntax error
            ......a................
a - "t.h", line 2.6: 1515-019 (S) Syntax is incorrect.
```

関連情報

- 「ユーザーズ・ガイド」の『**-d** オプション』
- 「ユーザーズ・ガイド」の『*C プリプロセッサーによる Fortran ファイルの引き渡し*』

PERMUTATION

PERMUTATION ディレクティブは、*integer_array_name_list* でリストされている各配列のエレメントが繰り返し値を持たないように指定します。このディレクティブは、配列エレメントが他の配列参照の添え字として使用されているときに便利です。

PERMUTATION ディレクティブが有効なのは、**-qsmp** または **-qhot** コンパイラー・オプションのいずれかが指定されているときに限られます。

構文

```
▸▸──PERMUTATION──(──integer_array_name_list──)──────────────────────────▸◂
```

integer_array_name
: 繰り返し値がない整数配列です。

規則

PERMUTATION ディレクティブに続く最初の非注釈行 (他のディレクティブは含まない) は、**DO** ループでなければなりません。この行は、無限の **DO** または **DO WHILE** ループにすることはできません。 **PERMUTATION** ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ネストされた **DO** ループには適用されません。

例

```
PROGRAM EX3
  INTEGER A(100), B(100)
  !SMP$  PERMUTATION (A)
  DO I = 1, 100
    A(I) = I
    B(A(I)) = B(A(I)) + A(I)
  END DO
END PROGRAM EX3
```

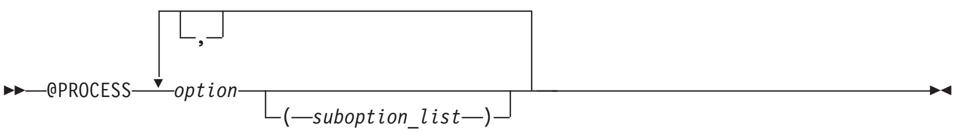
関連情報

- 「ユーザーズ・ガイド」の『**-qhot** オプション』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 335 ページの『DO』

@PROCESS

ソース・ファイルに **@PROCESS** コンパイラー・ディレクティブを入れることにより、個々のコンパイル単位に影響を与えるようにコンパイラー・オプションを指定することができます。構成ファイルまたはデフォルト設定あるいはコマンド行で指定したオプションをオーバーライドすることができます。

構文



option **-q** を持たないコンパイラー・オプションの名前です。

suboption
コンパイラー・オプションのサブオプションです。

規則

固定ソース形式では、**@PROCESS** は 1 桁目から、または 6 桁目より後から開始できます。自由ソース形式では、**@PROCESS** コンパイラー・ディレクティブは、どの桁からでも開始できます。

ステートメント・ラベルまたはインライン注釈を **@PROCESS** コンパイラー・ディレクティブと同じ行に入れることはできません。

デフォルト時には、**@PROCESS** コンパイラー・ディレクティブで指定するオプション設定は、ステートメントが存在するコンパイル単位に対してのみ有効です。ファイルが複数のコンパイル単位を持っている場合は、オプション設定は、次の単位がコンパイル

される前に、元の状態にリセットされます。 **DIRECTIVE** オプションによって指定されたトリガー定数は、ファイルの終わりまで (または、**NODIRECTIVE** が処理されるまで) 有効です。

@PROCESS コンパイラー・ディレクティブは、通常、コンパイル単位の最初のステートメントの前になければなりません。唯一の例外は、**SOURCE** および **NOSOURCE** を指定する場合です。この 2 つは、コンパイル単位内のいかなる場所にある **@PROCESS** ディレクティブでも使用できます。

関連情報

コンパイラー・オプションの詳細については、「ユーザーズ・ガイド」の『*コンパイラー・オプションの詳細*』を参照してください。

SNAPSHOT

SNAPSHOT ディレクティブを使用して、デバッグ・プログラムでブレークポイントを設定できる安全な位置を指定できます。さらに、デバッグ・プログラムに対して可視にする必要のある変数のセットを提供できます。 **SNAPSHOT** ディレクティブは、**-qsmp** コンパイラー・オプション (非マルチスレッド・プログラムでも使用できます) のサポートを提供します。

SNAPSHOT ディレクティブが設定されたポイントで、わずかなパフォーマンス低下がある場合があります。これは、デバッグ・プログラムがアクセスするために、変数をメモリーに保持する必要があるためです。 **SNAPSHOT** ディレクティブによって可視にされた変数は、読み取り専用です。これらの変数をデバッガーを通して変更すると、未定義の動作が発生します。慎重に使用してください。

構文

```
▶▶—SNAPSHOT—(—named_variable_list—)——————————————————————————▶◀
```

named_variable

現行の有効範囲でアクセス可能にする必要のある名前付き変数。

規則

SNAPSHOT ディレクティブを使用するには、コンパイル時に **-qdbg** コンパイラー・オプションを指定する必要があります。

例

例 1: 次の例では、プライベート変数の値をモニターするために、 **SNAPSHOT** ディレクティブが使用されています。

```
      INTEGER :: IDX
      INTEGER :: OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM
      INTEGER, ALLOCATABLE :: ARR(:)
!       ...
```

```
!$OMP PARALLEL, PRIVATE(IDX)
!$OMP MASTER
     ALLOCATE(ARR(OMP_GET_NUM_THREADS()))
!$OMP END MASTER
!$OMP BARRIER

    IDX = OMP_GET_THREAD_NUM() + 1

!IBM* SNAPSHOT(IDX)                  ! The PRIVATE variable IDX is made visible
                                     ! to the debugger.
    ARR(IDX) = 2*IDX + 1

!$OMP END PARALLEL
```

例 2: 次の例では、プログラムのデバッグで中間値をモニターするために、**SNAPSHOT** ディレクティブが使用されます。

```
      SUBROUTINE SHUFFLE(NTH, XDAT)
        INTEGER, INTENT(IN) :: NTH
        REAL, INTENT(INOUT) :: XDAT(:)
        INTEGER :: I_TH, IDX, PART(1), I, J, LB, UB
        INTEGER :: OMP_GET_THREAD_NUM
        INTEGER(8) :: Y=1
        REAL :: TEMP

        CALL OMP_SET_NUM_THREADS(NTH)
        PART = UBOUND(XDAT)/NTH

!$OMP   PARALLEL, PRIVATE(NUM_TH, I, J, LB, UB, IDX, TEMP), SHARED(XDAT)
          NUM_TH = OMP_GET_THREAD_NUM() + 1
          LB = (NUM_TH - 1)*PART(1) + 1
          UB = NUM_TH*PART(1)

          DO I=LB, UB
!$OMP       CRITICAL
              Y = MOD(65539_8*y, 2_8**31)
              IDX = INT(REAL(Y)/REAL(2_8**31)*(UB - LB) + LB)

!SMP$         SNAPSHOT(i, y, idx, num_th, lb, ub)

!$OMP       END CRITICAL
            TEMP = XDAT(I)
            XDAT(I) = XDAT(IDX)
            XDAT(IDX) = TEMP
         ENDDO

!SMP$    SNAPSHOT(TEMP)                    ! The user can examine the value
                                           ! of the TEMP variable

!$OMP  END PARALLEL
     END
```

関連情報

-qdbg コンパイラー・オプションの詳細については、「ユーザーズ・ガイド」を参照してください。

SOURCEFORM

SOURCEFORM コンパイラー・ディレクティブは、ファイルの終わりに到達するか、**@PROCESS** ディレクティブまたは別の **SOURCEFORM** ディレクティブが異なるソース形式を指定するまで、後続するすべての行を指定のソース形式で処理するように指示します。

構文

```
►►──SOURCEFORM──(──source──)──────────────────────────────────────────►◄
```

source 次のいずれかになります。 **FIXED**、 **FIXED**(*right_margin*)、 **FREE(F90)**、**FREE(IBM)**、または **FREE**。 **FREE** のデフォルト設定は、**FREE(F90)** です。

right_margin
右マージンの桁位置を指定する無符号の整数です。デフォルトは 72 桁です。最大では、132 桁になります。

規則

SOURCEFORM ディレクティブは、ファイルのどの位置にでも入れることができます。インクルード・ファイルは、インクルード・ファイルのソース形式でコンパイルされます。 **SOURCEFORM** ディレクティブがインクルード・ファイル内にある場合、インクルード・ファイルの処理が完了すると、ソース形式は、組み込み先のファイルのソース形式に戻ります。

SOURCEFORM ディレクティブでは、ラベルを指定することはできません。

ヒント

既存のファイルをインクルード・ファイルを含む Fortran 90 自由形式に変更するには、次のように行います。

1. インクルード・ファイルを Fortran 90 自由ソース形式に変換します。つまり、インクルード・ファイルの先頭に **SOURCEFORM** ディレクティブを追加します。たとえば、次のようにします。

   ```
   !CONVERT* SOURCEFORM (FREE(F90))
   ```

   この変換処理に対して *trigger_constant* を定義します。
2. すべてのインクルード・ファイルが変換されたら、.f ファイルを変換します。それぞれのファイルの先頭に同じ **SOURCEFORM** ディレクティブを追加するか、.f ファイルが **-qfree=f90** でコンパイルされることを確認します。
3. すべてのファイルが変換されたら、**-qnodirective** コンパイラー・オプションでディレクティブの処理を使用不能にします。コンパイル時に **-qfree=f90** が使用されているかを確認してください。不要な **SOURCEFORM** ディレクティブを削除することもできます。

例

```
@PROCESS DIRECTIVE(CONVERT*)
      PROGRAM MAIN          ! Main program not yet converted
      A=1; B=2
      INCLUDE 'freeform.f'
      PRINT *, RESULT       ! Reverts to fixed form
      END
```

ファイル freeform.f には以下のコードが含まれています。

```
!CONVERT* SOURCEFORM(FREE(F90))
RESULT = A + B
```

STREAM_UNROLL

STREAM_UNROLL ディレクティブはコンパイラーに対して、ソフトウェア・プリフェッチとループ・アンロールを結合した機能を反復カウントの大きな **DO** ループに適用することを指示します。ストリーム・アンロール機能は POWER4 以上のプラットフォームでのみ使用可能で、**DO** ループを最適化して複数のストリームを使用します。**STREAM_UNROLL** は内部と外部の両方の **DO** ループに指定できます。コンパイラーは、可能であればストリームの適切な番号を使用してストリーム・アンロールを実行します。依存関係を持つループに **STREAM_UNROLL** を適用すると、予期しない結果が生じます。

構文

```
►►─STREAM_UNROLL─┬───────────────────────┬──────────────────────────►◄
                 └─(─unroll_factor─)─┘
```

unroll_factor

unroll_factor は、1 以上の正整数初期化式でなければなりません。*unroll_factor* を 1 にするとループ・アンロールが使用不可になります。 *unroll_factor* を指定しないと、ストリーム・アンロールはコンパイラー依存になります。

規則

-qhot、**-qipa=level=2**、または **-qsmp** コンパイラー・オプションを指定して、ストリーム・アンロールを可能にする必要があります。最適化レベルを **-O4** 以上にすることによっても、コンパイラーでストリーム・アンロールを実行できます。

ストリーム・アンロールを行う場合、**STREAM_UNROLL** ディレクティブは **DO** ループの前になければなりません。

STREAM_UNROLL ディレクティブを複数回指定することはできません。また、1 つの **DO** 構文で、このディレクティブを **UNROLL**、**NOUNROLL**、**UNROLL_AND_FUSE**、または **NOUNROLL_AND_FUSE** ディレクティブと結合することはできません。

STREAM_UNROLL ディレクティブを **DO WHILE** ループ および無限 **DO** ループに指定することはできません。

例

以下に、パフォーマンスを向上できる **STREAM_UNROLL** の例を示します。

```
     integer, dimension(1000) :: a, b, c
     integer i, m, n

!IBM* stream_unroll(4)
      do i =1, n
        a(i) = b(i) + c(i)
      enddo
     end
```

unroll factor は、次のようにして、反復数を n から n/4 まで減らします。

```
m = n/4
do  i =1, n/4
    a(i) = b(i) + c(i)
    a(i+m) = b(i+m) + c(i+m)
    a(i+2*m) = b(i+2*m) + c(i+2*m)
    a(i+3*m) = b(i+3*m) + c(i+3*m)
enddo
```

読み取り操作と保管操作の増加数は、コンパイラーが決定したストリーム数に振り分けられ、計算時間が短縮されてパフォーマンスが向上します。

関連情報

- XL Fortran におけるプリフェッチ技法の使用の詳細については、**PREFETCH** ディレクティブを参照してください。
- ループ・アンロールを最適化する追加の方法については、**UNROLL** および **UNROLL_AND_FUSE** ディレクティブを参照してください。

SUBSCRIPTORDER

SUBSCRIPTORDER ディレクティブは、配列の添え字を再配置します。ディレクティブが宣言で配列次元の順序を変更するので、この結果、新しい配列形状になります。配列に対するすべての参照は、新しい配列形状に一致させるために、それに応じて再配置されます。

慎重に使用すれば、**SUBSCRIPTORDER** ディレクティブは、キャッシュ・ヒット数およびデータ・プリフェッチの量を増やすことによって、パフォーマンスを向上させることができます。パフォーマンスを最大限に上げる配置が見つかるまで、このディレクティブを試さなければならない場合があります。通常キャッシュを使用しないハードウェア・アーキテクチャー用であったコードを移植するときは、特に **SUBSCRIPTORDER** が便利です。

PowerPC などのキャッシュを使用するハードウェア・アーキテクチャーでは、各データ・エレメントにアクセスするために、データのキャッシュ・ライン全体がプロセッサーにロードされることがよくあります。ストレージ配置の変更を使用して、連続してアクセスされるエレメントを隣接して保管することができます。参照される各キャッシュ・ラインのエレメントのアクセスが増えるに従って、パフォーマンスが向上します。さらに、連続してアクセスされる配列エレメントを隣接して保管すると、プロセッサーのプリフェッチ機能を活用するために役立ちます。

構文

```
                     ┌─,─────────────────────┐
                     ▼                       │
►►──SUBSCRIPTORDER──(───subscriptorder_array─┴──)──────────────────────────►◄
```

subscriptorder_array の意味は次のとおりです。

```
              ┌─,────────────────────────┐
              ↓                          │
►►─array_name─(─subscriptorder_number─┴─)──────────────────────────►◄
```

array name
　　配列の名前です。

subscriptorder_number
　　整数定数です。

規則

SUBSCRIPTORDER ディレクティブは、subscriptorder_array リスト内の配列に対するすべての宣言または参照に先行する有効範囲単位で使用する必要があります。ディレクティブはその有効範囲単位のみに適用され、少なくとも 1 つの配列を含む必要があります。複数の有効範囲単位が 1 つの配列を共用する場合は、 **SUBSCRIPTORDER** ディレクティブを、同一の添え字配置で、適用できるそれぞれの有効範囲単位に適用する必要があります。有効範囲単位間で配列を共用する方法の例には、 **COMMON** ステートメント、**USE** ステートメント、およびサブルーチン引き数があります。

subscriptorder_number リスト内の最小の添え字の番号は、1 にする必要があります。最大番号は、対応する配列内の次元数と等しくなければなりません。これら 2 つの限界の間にある各整数の数値 (限界の値を含む) は、再配置の前の添え字の番号を示し、リストに正確に 1 度だけ含まれている必要があります。

有効範囲単位内の特定の配列に対し、**SUBSCRIPTORDER** ディレクティブを、複数回適用してはなりません。

配列の 1 つを **SUBSCRIPTORDER** ディレクティブで使用する場合、エレメント型プロシージャーに対する実際の引き数として配列を渡すときには、配列形状の一致を維持する必要があります。さらに、**SHAPE**、**SIZE**、**LBOUND**、および **UBOUND** 照会組み込みプロシージャーの実引き数、および大部分の変形可能組み込みプロシージャーの実引き数も調整する必要があります。

入力データ・ファイルのデータ、および **SUBSCRIPTORDER** ディレクティブ内で使用する配列の明示的初期化のデータは、手動で変更する必要があります。

さらに **COLLAPSE** ディレクティブも適用される配列上では、 **COLLAPSE** ディレクティブは、常に以前の subscriptorder 次元数を参照します。

想定サイズ配列の最後の次元を再配置してはなりません。

例

例 1: 次の例では、 **SUBSCRIPTORDER** ディレクティブは明示的形状配列に適用され、プログラム出力に影響を及ぼすことなく、配列のそれぞれの参照で添え字をスワップします。

```
!IBM* SUBSCRIPTORDER(A(2,1))
INTEGER COUNT/1/, A(3,2)

DO J = 1, 3
  DO K = 1, 2
    ! Inefficient coding: innermost index is accessing rightmost
    ! dimension.  The subscriptorder directive compensates by
    ! swapping the subscripts in the array's declaration and
    ! access statements.
    !
    A(J,K) = COUNT
    PRINT*, J, K, A(J,K)

    COUNT = COUNT + 1
  END DO
END DO
```

上記のディレクティブがない場合は、配列の形状は (3,2) になり、配列エレメントは、次の順序で保管されます。

```
A(1,1) A(2,1) A(3,1) A(1,2) A(2,2) A(3,2)
```

上記のディレクティブがある場合、配列の形状は (2,3) になり、配列エレメントは、次の順序で保管されます。

```
A(1,1) A(2,1) A(1,2) A(2,2) A(1,3) A(2,3)
```

関連情報

COLLAPSE ディレクティブの詳細については、515 ページの『COLLAPSE』を参照してください。

UNROLL

UNROLL ディレクティブはコンパイラーに対して、可能であればループ・アンロールを試みることを指示します。ループ・アンロールは **DO** ループの本体を複製して、ループを完了するために必要となる反復の回数を減らします。

-qunroll コンパイラー・オプションを指定して、ファイル全体のループ・アンロールを制御できます。特定の **DO** ループにディレクティブを指定すると、コンパイラー・オプションは必ずオーバーライドされます。

構文

```
►►─┬─UNROLL─┬───────────────────────┬─┬─────────────────────────────────►◄
   │        └─(─unroll_factor─)─────┘ │
   └─NOUNROLL─────────────────────────┘
```

unroll_factor

unroll_factor は、1 以上の正整数初期化式でなければなりません。*unroll_factor* を 1 にするとループ・アンロールが使用不可になります。 *unroll_factor* を指定しない場合、ループ・アンロールはコンパイラー依存になります。

規則

ループ・アンロールを行う場合、**UNROLL** ディレクティブは **DO** ループの前になければなりません。

UNROLL ディレクティブを複数回指定することはできません。また、1 つの **DO** 構文で、このディレクティブを **NOUNROLL**、**STREAM_UNROLL**、**UNROLL_AND_FUSE**、または **NOUNROLL_AND_FUSE** ディレクティブと結合することはできません。

UNROLL ディレクティブを **DO WHILE** ループ および無限 **DO** ループに指定することはできません。

例

例 1: この例では、1 回の繰り返しで 2 回の繰り返し作業が実行されるよう、ループ本体が複製可能であることをコンパイラーに通知するために、**UNROLL(2)** ディレクティブが使用されています。コンパイラーがループをアンロールすると、コンパイラーは 1000 回の繰り返しを実行するのではなく、500 回だけ繰り返しを実行します。

```
!IBM* UNROLL(2)
      DO I = 1, 1000
         A(I) = I
      END DO
```

コンパイラーが直前のループをアンロールすることを選択すると、コンパイラーはそのループが本質的に以下と同じになるようにそのループを変換します。

```
     DO I = 1, 1000, 2
        A(I) = I
        A(I+1) = I + 1
     END DO
```

例 2: 最初の **DO** ループでは、 **UNROLL(3)** が使用されています。アンロールが実行されると、コンパイラーは 1 回の繰り返しで 3 回の繰り返し作業が行われるように、ループをアンロールします。 2 番目の **DO** ループでは、コンパイラーはパフォーマンスが最大になるようにループをアンロールする方法を決定します。

```
      PROGRAM GOODUNROLL

      INTEGER I, X(1000)
      REAL A, B, C, TEMP, Y(1000)

!IBM* UNROLL(3)
      DO I = 1, 1000
```

```
      X(I) = X(I) + 1
   END DO

!IBM* UNROLL
   DO I = 1, 1000
      A = -I
      B = I + 1
      C = I + 2
      TEMP = SQRT(B*B - 4*A*C)
      Y(I) = (-B + TEMP) / (2*A)
   END DO
   END PROGRAM GOODUNROLL
```

関連情報

- ループ・アンロールを最適化する追加の方法については、 530 ページの『STREAM_UNROLL』ディレクティブおよび『UNROLL_AND_FUSE』ディレクティブに記載されています。

UNROLL_AND_FUSE

UNROLL_AND_FUSE ディレクティブはコンパイラーに対して、可能であればループ・アンロールとヒューズを試みることを指示します。ループ・アンロールは、**DO** ループの複数の本体を複製して、必要な反復を 1 つのアンロール・ループに結合します。フューズ・ループを使用すると、ループ反復の必要数を最小化し、その上、キャッシュ・ミスの回数を減らすことができます。依存関係を持つループに **UNROLL_AND_FUSE** を適用すると、予期しない結果が生じます。

構文

```
>>─┬─UNROLL_AND_FUSE─┬──────────────────────┬─┬──────────────────────><
   │                 └─(─unroll_factor─)─┘ │
   └─NOUNROLL_AND_FUSE──────────────────────┘
```

unroll_factor

unroll_factor は、1 以上の正整数初期化式でなければなりません。*unroll_factor* を 1 にするとループ・アンロールが使用不可になります。 *unroll_factor* を指定しない場合、ループ・アンロールはコンパイラー依存になります。

規則

ループ・アンロールを行う場合、**UNROLL_AND_FUSE** ディレクティブは **DO** ループの前になければなりません。

UNROLL_AND_FUSE ディレクティブを最も内側の **DO** ループに指定することはできません。

UNROLL_AND_FUSE ディレクティブを複数回指定することはできません。また、1 つの **DO** の構文で、このディレクティブを **NOUNROLL_AND_FUSE**、**NOUNROLL**、**UNROLL**、または **STREAM_UNROLL** ディレクティブと結合することはできません。

UNROLL_AND_FUSE ディレクティブを **DO WHILE** ループ および無限 **DO** ループに指定することはできません。

例

例 1: 次の例では、**UNROLL_AND_FUSE** ディレクティブがループの本体を複製してフューズします。これにより、配列 *B* のキャッシュ・ミスの数が減少します。

```
      INTEGER, DIMENSION(1000, 1000) :: A, B, C
!IBM* UNROLL_AND_FUSE(2)
      DO I = 1, 1000
         DO J = 1, 1000
            A(J,I) = B(I,J) * C(J,I)
         END DO
      END DO
      END
```

下の **DO** ループは、**UNROLL_AND_FUSE** ディレクティブを適用した場合の、考えられる結果を示します。

```
    DO I = 1, 1000, 2
         DO J = 1, 1000
            A(J,I) = B(I,J) * C(J,I)
            A(J,I+1) = B(I+1, J) * C(J, I+1)
         END DO
      END DO
```

例 2: 以下の例では、複数の **UNROLL_AND_FUSE** ディレクティブを使用します。

```
      INTEGER, DIMENSION(1000, 1000) :: A, B, C, D, H
!IBM* UNROLL_AND_FUSE(4)
      DO I = 1, 1000
!IBM* UNROLL_AND_FUSE(2)
         DO J = 1, 1000
            DO k = 1, 1000
               A(J,I) = B(I,J) * C(J,I) + D(J,K)*H(I,K)
            END DO
         END DO
      END DO
      END
```

関連情報

- ループ・アンロールを最適化する追加の方法については、 530 ページの『STREAM_UNROLL』ディレクティブおよび 534 ページの『UNROLL』ディレクティブに記載されています。

IBM 拡張 の終り

第 12 章 組み込みプロシージャー

FORTRAN では、任意のプログラムで使用できる、組み込みプロシージャーと呼ばれるプロシージャーが定義されています。この章では、これらのプロシージャーをアルファベット順に解説します。

関連情報:

1. 191 ページの『組み込みプロシージャー』には、この章を読み進む前に理解しておく必要のある背景情報が記載されています。
2. 409 ページの『INTRINSIC』は関連記述です。

組み込みプロシージャーのクラス

組み込みプロシージャーには、照会関数、エレメント型プロシージャー、システム照会関数、変換関数、およびサブルーチンの 5 つのクラスがあります。

照会組み込み関数

照会関数 の結果は、その引き数の値ではなく、その主引き数の特性によって決まります。引き数の値は、定義する必要はありません。

ALLOCATED	LEN	RADIX
ASSOCIATED	LOC **1**	RANGE
BIT_SIZE	MAXEXPONENT	SHAPE
DIGITS	MINEXPONENT	SIZE
EPSILON	NUM_PARTHDS **1**	SIZEOF **1**
HUGE	NUM_USRTHDS **1**	TINY
KIND	PRECISION	UBOUND
LBOUND	PRESENT	

注:

1. IBM 拡張

エレメント型組み込みプロシージャー

組み込み関数の中のいくつかと、1 つの組み込みサブルーチン (**MVBITS**) はエレメント型 *(elemental)* です。つまり、これらはスカラー引き数に対して指定することができますが、配列である引き数も受け入れます。

すべての引き数がスカラーである場合は、結果はスカラーになります。

任意の引き数が配列である場合は、引き数 INTENT(OUT) および INTENT(INOUT) はすべて同じ型の配列でなければならず、その他の引き数は、この 2 つの引き数と適合しなければなりません。

結果の形状は、最高のランクを持つ引き数の形状になります。結果のエレメントは、各引き数の対応するエレメントに関数が個々に適用された場合と同じになります。

- ABS
- ACHAR
- ACOS
- ACOSD [1]
- ADJUSTL
- ADJUSTR
- AIMAG
- AINT
- ANINT
- ASIN
- ASIND [1]
- ATAN
- ATAND [1]
- ATAN2
- ATAN2D [1]
- BTEST
- CEILING
- CHAR
- CMPLX
- CONJG
- COS
- COSD [1]
- COSH
- CVMGx [1]
- DBLE
- DCMPLX [1]
- DIM
- DPROD
- ERF [1]
- ERFC [1]
- EXP
- EXPONENT
- FLOOR
- FRACTION
- GAMMA [1]
- HFIX [1]
- IACHAR
- IAND
- IBCLR
- IBITS
- IBSET
- ICHAR
- IEOR
- ILEN [1]
- INDEX
- INT
- INT2 [1]
- IOR
- ISHFT
- ISHFTC
- LEADZ [1]
- LEN_TRIM
- LGAMMA [1]
- LGE
- LGT
- LLE
- LLT
- LOG
- LOG10
- LOGICAL
- LSHIFT [1]
- MAX
- MERGE
- MIN
- MOD
- MODULO
- MVBITS
- NEAREST
- NINT
- NOT
- QCMPLX [1]
- QEXT [1]
- REAL
- RRSPACING
- RSHIFT
- SCALE
- SCAN
- SET_EXPONENT
- SIGN
- SIN
- SIND [1]
- SINH
- SPACING
- SQRT
- TAN
- TAND [1]
- TANH
- VERIFY

注:

1. IBM 拡張

システム照会組み込み関数

IBM 拡張

システム照会関数 は、制限式の中で使用できます。システム照会関数は、初期化式の中で使用できません。また、実引き数として渡すこともできません。

NUMBER_OF_PROCESSORS
PROCESSORS_SHAPE

IBM 拡張 の終り

変換組み込み関数

その他の組み込み関数はすべて、*変換関数* として分類されます。通常、これらの関数は配列引き数を受け入れて、配列の結果を戻します。配列の結果は、引き数配列内のエレメントによって異なります。

ALL	MAXVAL	SELECTED_INT_KIND
ANY	MINLOC	SELECTED_REAL_KIND
COUNT	MINVAL	SPREAD
CSHIFT	NULL 1	SUM
DOT_PRODUCT	PACK	TRANSFER
EOSHIFT	PRODUCT	TRANSPOSE
MATMUL	REPEAT	TRIM
MAXLOC	RESHAPE	UNPACK

注:

1. Fortran 95

配列に関する背景情報については、 83 ページの『第 4 章 配列の概念』 を参照してください。

組み込みサブルーチン

組み込みプロシージャーの中には、サブルーチンのものもあります。これらは、さまざまなタスクを実行します。

ABORT 1	MVBITS	SRAND 1
CPU_TIME 2	RANDOM_NUMBER	SYSTEM 1
DATE_AND_TIME	RANDOM_SEED	SYSTEM_CLOCK
GETENV 1	SIGNAL 1	

注:

1. IBM 拡張
2. Fortran 95

データ表示モデル

整数ビット・モデル

次のモデルでは、プロセッサーが負でないスカラー整数オブジェクトの各ビットをどのように表示するかを示しています。

$$j = \sum_{k=0}^{s-1} w_k \times 2^k$$

j 整数値です。

s ビット数です。

w_k 位置 *k* にある 2 進数の桁 *w* です。

IBM 拡張

XL Fortran は、XL Fortran 整数 kind 型付きパラメーターに対して、次の *s* パラメーターをインプリメントしています。

整数 Kind パラメーター	*s* パラメーター
1	8
2	16
4	32
8	64

IBM 拡張 の終り

次の組み込み関数がこのモデルを使用しています。

BTEST	IBSET	ISHFTC
IAND	IEOR	MVBITS
IBCLR	IOR	NOT
IBITS	ISHFT	

整数データ・モデル

$$i = s \times \sum_{k=1}^{q} w_k \times r^{k-1}$$

i 整数値です。

s 符号 (±1) です。

q 桁数 (正の整数) です。

w_k *r* より小さい、負でない数字です。

r 基数です。

IBM 拡張

XL Fortran は、次の *r* および *q* パラメーターを持つこのモデルをインプリメントしています。

整数 Kind パラメーター	*r* パラメーター	*q* パラメーター
1	2	7
2	2	15
4	2	31
8	2	63

IBM 拡張 の終り

次の組み込み関数がこのモデルを使用しています。

DIGITS　RADIX　RANGE
HUGE

実データ・モデル

$$x = \begin{cases} 0 & \text{または} \\ s \times b^{e} \times \sum_{k=1}^{p} f_k \times b^{-k} & \end{cases}$$

x 実数の値です。

s 符号 (±1) です。

b 1 より大きな整数です。

e 整数で、$e_{min} \le e \le e_{max}$ の関係にあります。

p 1 より大きな整数です。

f_k *b* より小さい、負でない整数 ($f_1 \ne 0$) です。

注: x=0 であれば、e=0 で、かつすべての f_k=0 です。

IBM 拡張

XL Fortran は、次のパラメーターを持つこのモデルをインプリメントしています。

実数 Kind パラメーター	*b* パラメーター	*p* パラメーター	e_{min} パラメーター	e_{max} パラメーター
4	2	24	-125	128
8	2	53	-1021	1024
16	2	106	-1021	1024

IBM 拡張 の終り

次の組み込み関数がこのモデルを使用しています。

DIGITS
EPSILON
EXPONENT
FRACTION
HUGE
MAXEXPONENT
MINEXPONENT
NEAREST
PRECISION
RADIX
RANGE
RRSPACING
SCALE
SET_EXPONENT
SPACING
TINY

組み込みプロシージャーの詳しい記述

次に示すのは、組み込みプロシージャーのすべてを総称名のアルファベット順リストです。

個々のプロシージャーに対して、いくつかの項目情報を記載します。

注:

1. 表題にリストされる引き数名は、プロシージャーを呼び出すときにキーワード引き数の名前として使用できます。
2. 特定名を持つそれらのプロシージャーに関して、表には特定の関数の説明とともに個々の特定名がリストされます。
   - 関数戻りタイプまたは引き数タイプが小文字で示されている場合は、そのタイプが小文字で指定されていることを示しますが、コンパイラーは、 **-qintsize**、 **-qrealsize**、 **-qautodbl** オプションの設定次第で、実際には特定名への呼び出しを置き換えることができます。

     たとえば、**SINH** への参照は、**-qrealsize=8** が有効な場合には、**DSINH** への参照に置き換えられ、**DSINH** への参照は **QSINH** への参照に置き換えられます。
   - 「引き数渡し」と書かれている欄は、その特定名をプロシージャーの実引き数として渡すことができるかどうかを示します。組み込みプロシージャーでは、特定名だけを引き数として渡すことができます。ただし、いくつかの特定名だけに限ります。このようにして渡された特定名は、スカラー引き数でのみ参照することができます。
3. 特定名はわかっていても、総称名がわからない場合は、指標で個々の特定名によって調べることができます。

ABORT ()

IBM 拡張

プログラムを終了します。オープンしているすべての出力ファイルをファイル・ポインターの現在位置に切り捨て、オープンしているすべてのファイルをクローズし、SIGIOT シグナルを現行プロセスに送信します。

SIGIOT が受信も無視もされず、現行ディレクトリーが書き込み可能な場合は、システムは現行ディレクトリーにコア・ファイルを作成します。

クラス

サブルーチン

例

ABORT サブルーチンを使用するステートメントの例を以下に示します。

```
IF (ERROR_CONDITION) CALL ABORT
```

上記のプログラムによって生成される出力結果は次のとおりです。

```
 /home/mark
IOT/Abort trap(coredump)
```

IBM 拡張 の終り

ABS (A)

絶対値を求めます。

A タイプは整数、実数、複素数のいずれかでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

A が複素数の場合以外、 A と同じです。結果は必ず実数になります。

結果値

- A のタイプが整数または実数の場合は、結果は |A| になります。
- A のタイプが値 (x,y) を持つ複素数の場合は、結果は以下に近似します。

$$\sqrt{x^2 + y^2}$$

例

ABS ((3.0, 4.0)) は値 5.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
IABS	任意の整数 **2**	引き数と同じ	あり
ABS	デフォルトの実数	デフォルトの実数	あり
DABS	倍精度実数	倍精度実数	あり
QABS **1**	REAL(16)	REAL(16)	あり
CABS	デフォルトの複素数	デフォルトの実数	あり

特定名	引き数タイプ	結果タイプ	引き数渡し
CDABS 1	倍精度複素数	倍精度実数	あり
ZABS 1	倍精度複素数	倍精度実数	あり
CQABS 1	COMPLEX(16)	REAL(16)	あり

注:

1. IBM 拡張
2. IBM 拡張: デフォルト以外の整数の引き数を指定するための機能。

ACHAR (I)

ASCII 照合順序の指定された位置に文字を戻します。これは、IACHAR 関数の反対です。

引き数タイプおよび属性

I タイプは整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

KIND ('A') と同じ kind 型付きパラメーターを持つ長さ 1 の文字です。

結果値

- I が 0 ≤ I ≤ 127 の範囲内の値を持っている場合は、結果は ASCII 照合順序の位置 I にある文字になります。ただし、これは I に対応する文字が表示可能な場合です。
- I が許可されている値の範囲外にある場合は、結果は不定になります。

例

ACHAR (88) は値 'X' を持ちます。

ACOS (X)

アークコサイン (逆余弦) 関数です。

引き数タイプおよび属性

X タイプは実数でなければなりません。値は不等式 |X| ≤ 1 を満たさなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- ラジアンで表され、arccos(X) とほぼ同じ値になります。
- 0 ≤ ACOS(X) ≤ π の範囲内にあります。

例

ACOS (1.0) は値 0.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ACOS	デフォルトの実数	デフォルトの実数	あり
DACOS	倍精度実数	倍精度実数	あり
QACOS 1	REAL(16)	REAL(16)	あり
QARCOS 1	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張

ACOSD (X)

IBM 拡張

アークコサイン (逆余弦) 関数です。 結果は「度」の単位になります。

引き数タイプおよび属性

X タイプは実数でなければなりません。 値は不等式 |X| ≤ 1 を満たさなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 「度」の単位で表され、arccos(X) とほぼ同じ値になります。
- 0° ≤ ACOSD(X) ≤ 180° の範囲内にあります。

例

ACOSD (0.5) は値 60.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ACOSD	デフォルトの実数	デフォルトの実数	あり
DACOSD	倍精度実数	倍精度実数	あり
QACOSD	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

ADJUSTL (STRING)

先行ブランクを除去し、後続ブランクを挿入して左にそろえます。

引き数タイプおよび属性

STRING タイプは文字でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

STRING と同じ長さと kind 型付きパラメーターの文字。

結果値

結果の値は、先行ブランクが削除されて、同数の後続ブランクが挿入されること以外は STRING と同じになります。

例

ADJUSTL ('ƀWORD') は、値 'WORDƀ' になります。

ADJUSTR (STRING)

後続ブランクを除去し、先行ブランクを挿入して右にそろえます。

引き数タイプおよび属性

STRING タイプは文字でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

STRING と同じ長さと kind 型付きパラメーターの文字。

結果値

結果の値は、後続ブランクが削除されて、同数の先行ブランクが挿入されること以外は STRING と同じです。

例

ADJUSTR ('WORDƀ') は、値 'ƀWORD' になります。

AIMAG (Z), IMAG (Z)

複素数の虚数部分

引き数タイプおよび属性

Z タイプは複素数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

Z と同じ kind 型付きパラメーターを持つ実数

結果値

Z が値 (x,y) を持っている場合は、結果は値 y になります。

例

AIMAG ((2.0, 3.0)) は値 3.0 になります。

特定名	引き数タイプ	結果タイプ	引き数渡し
AIMAG	デフォルトの複素数	デフォルトの実数	あり
DIMAG 1	倍精度複素数	倍精度実数	あり

特定名	引き数タイプ	結果タイプ	引き数渡し
QIMAG 1	COMPLEX(16)	REAL(16)	あり

注:

1. IBM 拡張

AINT (A, KIND)

整数に切り捨てます。

引き数タイプおよび属性

A タイプは実数でなければなりません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 結果のタイプは実数です。
- KIND が存在する場合は、 kind 型付きパラメーターは KIND で指定されたタイプになり、それ以外の場合は、A のタイプになります。

結果値

- $|A| < 1$ の場合は、結果はゼロになります。
- $|A| \geq 1$ の場合は、結果は絶対値が A の絶対値を超えない最大整数に等しい値を持ち、符号は A の符号と同じです。

例

```
AINT(3.555) = 3.0
AINT(-3.555) = -3.0
```

特定名	引き数タイプ	結果タイプ	引き数渡し
AINT	デフォルトの実数	デフォルトの実数	あり
DINT	倍精度実数	倍精度実数	あり
QINT 1	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張

ALL (MASK, DIM)

配列全体内のすべての値、または単一次元に沿った個々のベクトル内のすべての値が真であるかどうかを判別します。

引き数タイプおよび属性

MASK　論理配列です。

DIM (オプション)
: 1 ≤ DIM ≤ rank(MASK) の範囲内の整数スカラーです。対応する実引き数は、オプションの仮引き数であってはなりません。

クラス

変換関数

結果値

結果は、MASK と同じタイプおよび型付きパラメーターを持つ論理配列で、ランクは rank(MASK)-1 になります。 DIM を省略した場合、または MASK のランクが 1 の場合は、結果は論理タイプのスカラーになります。

結果の形状は、(s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) で、この n は MASK のランクです。

結果配列の中の個々のエレメントが .TRUE. になるのは、 MASK(m_1, m_2, ..., $m_{(DIM-1)}$, :, $m_{(DIM+1)}$, ..., m_n) で指定されたすべてのエレメントが真である場合だけです。結果がスカラーである場合は、DIM が指定されていないか、または MASK のランクが 1 であるという理由で、.TRUE. になります。しかし、これは MASK のすべてのエレメントが真であるか、または MASK のサイズがゼロの場合に限ります。

例

```
! A is the array | 4 3 6 |, and B is the array | 3 5 2 |
!                | 2 4 1 |                     | 7 8 4 |

! Is every element in A less than the
! corresponding one in B?
      RES = ALL(A .LT. B)          ! result RES is false

! Are all elements in each column of A less than the
! corresponding column of B?
      RES = ALL(A .LT. B, DIM = 1) ! result RES is (f,t,f)
```

```
! Same question, but for each row of A and B.
      RES = ALL(A .LT. B, DIM = 2) ! result RES is (f,t)
```

ALLOCATED(ARRAY) または ALLOCATED(SCALAR)

割り振り可能オブジェクトが現在割り振られているかどうかを示します。

引き数タイプおよび属性

ARRAY 割り振り状況を知りたい割り振り可能配列です。

SCALAR 割り振り状況を知りたい割り振り可能スカラーです。

クラス

照会関数

結果タイプおよび属性

デフォルトの論理スカラー

結果値

結果は、ARRAY または SCALAR の割り振り状況に対応します。つまり、現在割り振られている場合は .TRUE.、現在割り振られていない場合は .FALSE.、割り振り状況が未定義の場合は undefined になります。 **-qxlf90=autodealloc** コンパイラー・オプションを使用してコンパイルする場合、未定義の割り振り状況はありません。

例

```
INTEGER, ALLOCATABLE, DIMENSION(:) :: A
PRINT *, ALLOCATED(A)      ! A is not allocated yet.
ALLOCATE (A(1000))
PRINT *, ALLOCATED(A)      ! A is now allocated.
END
```

関連情報

90 ページの『割り振り可能配列』、 290 ページの『ALLOCATE』、 77 ページの『割り振り状況』を参照してください。

ANINT (A, KIND)

最も近い整数を求めます。

引き数タイプおよび属性

A タイプは実数でなければなりません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 結果のタイプは実数です。
- KIND が存在する場合は、 kind 型付きパラメーターは KIND で指定されたタイプになり、それ以外の場合は、A のタイプになります。

結果値

- A > 0 である場合は、ANINT(A) = AINT(A + 0.5)
- A ≤ 0 である場合は、ANINT(A) = AINT(A - 0.5)

注: 0.5 の加算と減算は、ゼロへの丸めモードで実行されます。

例

```
ANINT(3.555) = 4.0
ANINT(-3.555) = -4.0
```

特定名	引き数タイプ	結果タイプ	引き数渡し
ANINT	デフォルトの実数	デフォルトの実数	あり
DNINT	倍精度実数	倍精度実数	あり
QNINT 1	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張

ANY (MASK, DIM)

配列全体内の値のいずれか、または単一次元に沿った個々のベクトル内のいずれかの値が真であるかどうかを判別します。

引き数タイプおよび属性

MASK
論理配列です。

DIM (オプション)
1 ≤ DIM ≤ rank(MASK) の範囲内の整数スカラーです。対応する実引き数は、オプションの仮引き数であってはなりません。

クラス

変換関数

結果値

結果は、MASK と同じタイプおよび型付きパラメーターの論理配列で、ランクは rank(MASK)-1 になります。 DIM が脱落している場合、または MASK のランクが 1 の場合は、結果は論理タイプのスカラーになります。

結果の形状は、(s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) で、この n は MASK のランクです。

結果配列の中の個々のエレメントが .TRUE. になるのは、 MASK(m_1, m_2, ..., $m_{(DIM-1)}$, :, $m_{(DIM+1)}$, ..., m_n) で指定されたどのエレメントも真である場合だけです。結果がスカラーである場合は、DIM が指定されていないか、または MASK のランクが 1 であるという理由で、.TRUE. になります。しかし、これは MASK のエレメントのいずれかが真である場合に限ります。

例

```
! A is the array | 9 -6 7 |, and B is the array | 2 7 8 |
!                | 3 -1 5 |                     | 5 6 9 |

! Is any element in A greater than or equal to the
! corresponding element in B?
      RES = ANY(A .GE. B)          ! result RES is true

! For each column in A, is there any element in the column
! greater than or equal to the corresponding element in B?
      RES = ANY(A .GE. B, DIM = 1) ! result RES is (t,f,f)

! Same question, but for each row of A and B.
      RES = ANY(A .GE. B, DIM = 2) ! result RES is (t,f)
```

ASIN (X)

アークサイン (逆正弦) 関数です。

引き数タイプおよび属性

X タイプは実数でなければなりません。 値は不等式 |X| ≤ 1 を満たさなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- ラジアンで表され、arcsin(X) とほぼ同じ値になります。
- $-\pi/2 \le$ ASIN(X) $\le \pi/2$ の範囲内にあります。

例

ASIN (1.0) は $\pi/2$ に近似します。

特定名	引き数タイプ	結果タイプ	引き数渡し
ASIN	デフォルトの実数	デフォルトの実数	あり
DASIN	倍精度実数	倍精度実数	あり
QASIN **1**	REAL(16)	REAL(16)	あり
QARSIN **1**	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張

ASIND (X)

IBM 拡張

アークサイン (逆正弦) 関数です。結果は「度」の単位になります。

引き数タイプおよび属性

X タイプは実数でなければなりません。 値は不等式 $|X| \le 1$ を満たさなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 「度」の単位で表され、arcsin(X) とほぼ同じ値になります。
- $-90° \le$ ASIND(X) $\le 90°$ の範囲内にあります。

例

ASIND (0.5) は値 30.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ASIND	デフォルトの実数	デフォルトの実数	あり
DASIND	倍精度実数	倍精度実数	あり
QASIND	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

ASSOCIATED (POINTER, TARGET)

そのポインター引き数の関連付け状況を戻すか、または、ポインターがターゲットと関連付けられているかどうかを示します。

引き数タイプおよび属性

POINTER 関連付け状況をテストしたいポインターです。タイプはどんなタイプでもかまいません。関連付け状況は未定義であってはなりません。

TARGET (オプション)
POINTER と関連付けられているか、または関連付けられていないポインターまたはターゲットです。関連付け状況は未定義であってはなりません。

クラス

照会関数

結果タイプおよび属性

デフォルトの論理スカラー

結果値

POINTER 引き数だけが指定されているとき、いずれかのターゲットと関連する場合は .TRUE. になり、それ以外の場合は .FALSE. になります。 TARGET も指定されている場合は、POINTER が TARGET と関連しているかどうか、あるいは、TARGET が関連しているのと同じオブジェクトと関連しているかどうか (TARGET がポインターでもある場合) をプロシージャーはテストします。

POINTER または TARGET がサイズがゼロ配列と関連している場合、あるいは、TARGET のサイズがゼロ配列の場合には、結果は不定になります。

異なるタイプまたは形状を持つオブジェクトは、互いに関連させることはできません。

タイプと形状が同じで境界が異なる配列は、互いに関連させることができます。

例

```
REAL, POINTER, DIMENSION(:,:) :: A
REAL, TARGET, DIMENSION(5,10) :: B, C

NULLIFY (A)
PRINT *, ASSOCIATED (A)   ! False, not associated yet

A => B
PRINT *, ASSOCIATED (A)   ! True, because A is
                          ! associated with B

PRINT *, ASSOCIATED (A,C) ! False, A is not
                          ! associated with C
END
```

ATAN (X)

アークタンジェント (逆正接) 関数です。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- ラジアンで表され、arctan(X) とほぼ同じ値になります。
- $-\pi/2 \leq$ ATAN(X) $\leq \pi/2$ の範囲内です。

例

ATAN (1.0) は $\pi/4$ に近似します。

特定名	引き数タイプ	結果タイプ	引き数渡し
ATAN	デフォルトの実数	デフォルトの実数	あり
DATAN	倍精度実数	倍精度実数	あり
QATAN 1	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張

ATAND (X)

IBM 拡張

アークタンジェント (逆正接) 関数です。結果は「度」の単位になります。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 「度」の単位で表され、arctan(X) とほぼ同じ値になります。
- -90° ≤ ATAND(X) ≤ 90° の範囲内にあります。

例

ATAND (1.0) は値 45.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ATAND	デフォルトの実数	デフォルトの実数	あり
DATAND	倍精度実数	倍精度実数	あり
QATAND	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

ATAN2 (Y, X)

アークタンジェント (逆正接) 関数です。結果は実引き数 Y と X で形成されているゼロ以外の複素数 (X,Y) のプリンシパル値です。

引き数タイプおよび属性

Y タイプは実数でなければなりません。

X タイプおよび kind 型付きパラメーターは Y と同じでなければなりません。Y が値ゼロを持っている場合は、X は値ゼロを持ってはなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- ラジアンで表され、複素数 (X, Y) の引き数のプリンシパル値に等しい値を持ちます。
- $-\pi < \text{ATAN2(Y, X)} \le \pi$ の範囲内にあります。
- $X \neq 0$ の場合は、結果は arctan(Y/X) とほぼ同じ値になります。
- $Y > 0$ の場合は、結果は正になります。
- $Y < 0$ の場合は、結果は負になります。
- $Y = 0$ で $X > 0$ の場合は、結果はゼロになります。
- $Y = 0$ で $X < 0$ の場合は、結果は π になります。
- $X = 0$ の場合は、結果の絶対値は $\pi/2$ になります。

例

ATAN2 (1.5574077, 1.0) は値 1.0 を持ちます。

次のように設定すると、

```
Y = |  1   1 |          X = | -1  1 |
    | -1  -1 |              | -1  1 |
```

ATAN2(Y,X) の値は、およそ以下のようになります。

```
ATAN2 (Y, X) = |  3π/4   π/4 |
               | -3π/4  -π/4 |
```

特定名	引き数タイプ	結果タイプ	引き数渡し
ATAN2	デフォルトの実数	デフォルトの実数	あり
DATAN2	倍精度実数	倍精度実数	あり
QATAN2 1	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張

ATAN2D (Y, X)

IBM 拡張

アークタンジェント (逆正接) 関数です。結果は実引き数 Y と X で形成されているゼロ以外の複素数 (X,Y) のプリンシパル値です。

引き数タイプおよび属性

Y タイプは実数でなければなりません。

X タイプおよび kind 型付きパラメーターは Y と同じでなければなりません。Y が値ゼロを持っている場合は、X は値ゼロを持ってはなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 「度」の単位で表され、複素数 (X, Y) の引き数のプリンシパル値に等しい値を持ちます。
- -180° < ATAN2D(Y,X) ≤ 180° の範囲内にあります。
- X≠0 の場合は、結果は arctan(Y/X) とほぼ同じ値になります。
- Y>0 の場合は、結果は正になります。
- Y<0 の場合は、結果は負になります。
- Y=0 で X>0 の場合は、結果はゼロになります。
- Y=0 で X<0 の場合は、結果は 180° になります。
- X=0 の場合は、結果の絶対値は 90° になります。

例

ATAN2D (1.5574077, 1.0) は値 57.295780181 (近似値) を持ちます。

次のように設定すると、

```
Y = |  1.0   1.0 |      X = | -1.0   1.0 |
    | -1.0  -1.0 |          | -1.0   1.0 |
```

ATAN2D(Y,X) の値は、以下のようになります。

```
ATAN2D(Y,X) =  |  135.0000000   45.00000000 |
               | -135.0000000  -45.00000000 |
```

特定名	引き数タイプ	結果タイプ	引き数渡し
ATAN2D	デフォルトの実数	デフォルトの実数	あり
DATAN2D	倍精度実数	倍精度実数	あり
QATAN2D	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

BIT_SIZE (I)

ビット数を整数タイプで戻します。検査されるのは引き数のタイプだけなので、引き数を定義する必要はありません。

引き数タイプおよび属性

I タイプは整数でなければなりません。

クラス

照会関数

結果タイプおよび属性

I と同じ kind 型付きパラメーターを持つスカラー整数です。

結果値

結果は、引き数の整数データ型の中のビットの数になります。

IBM 拡張

```
   type                  bits
-----------             ------
 integer(1)                08
 integer(2)                16
 integer(4)                32
 integer(8)                64
```

IBM 拡張 の終り

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

kind 4 の整数タイプ (つまり、4 バイトの整数) には 32 のビットが含まれているので、BIT_SIZE (1_4) は値 32 を持ちます。

BTEST (I, POS)

整数値のビットをテストします。

引き数タイプおよび属性

I タイプは整数でなければなりません。

POS タイプは整数でなければなりません。 非負数で、BIT_SIZE (I) よりも小さくなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

結果のタイプは、デフォルトの論理値になります。

結果値

I のビット POS が値 1 を持っている場合、結果は値 .TRUE. を持ち、I のビット POS が値ゼロを持っている場合は、値 .FALSE. を持ちます。

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

BTEST (8, 3) は値 .TRUE. を持ちます。

```
If A has the value
   | 1 2 |
   | 3 4 |
the value of BTEST (A, 2) is
   | false false |
   | false true  |
and the value of BTEST (2, A) is
   | true  false |
   | false false |
```

542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
BTEST **1**	任意の整数	デフォルトの論理値	あり

注:

1. IBM 拡張

CEILING(A, KIND)

その引き数よりも大きいかまたは等しい最小整数を戻します。

引き数タイプおよび属性

A タイプは実数でなければなりません。

Fortran 95

KIND (オプション)
スカラー整数の初期化式でなければなりません。

Fortran 95 の終り

クラス

エレメント型関数

結果タイプおよび属性

- タイプは整数です。

Fortran 95

- KIND が存在する場合は、kind 型付きパラメーターは KIND で指定されたものになります。存在しない場合は、KIND 型付きパラメーターはデフォルトの整数タイプになります。

Fortran 95 の終り

結果値

結果は、A 以上の最小整数に等しい値を持ちます。

Fortran 95

指定された KIND の整数として結果を表現できない場合は、この結果は未定義になります。

Fortran 95 の終り

例

```
CEILING(-3.7) は値 -3 を持ちます。
CEILING(3.7) は値 4 を持ちます。
```

Fortran 95

`CEILING(1000.1, KIND=2)` は値 1 001 と kind 型付きパラメーター 2 を持ちます。

Fortran 95 の終り

CHAR (I, KIND)

指定された kind 型付きパラメーターと関連した照合順序の所定の位置にある文字を戻します。これは、関数 ICHAR の逆です。

引き数タイプおよび属性

IBM 拡張

I 値が 0 ≤ I ≤ 127 の範囲内の整数タイプでなければなりません。

IBM 拡張 の終り

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 長さ 1 の文字
- KIND が存在する場合は、 kind 型付きパラメーターは KIND で指定されたタイプになり、それ以外の場合は、デフォルトの文字タイプになります。

結果値

- 結果は、指定された kind 型付きパラメーターと関連した照合順序の位置 I にある文字になります。
- ICHAR (CHAR (I, KIND (C))) は、 0 ≤ I ≤ 127 に対しては値 I を持たなければならず、CHAR (ICHAR (C), KIND (C)) は、任意の表示可能文字に対しては値 C を持たなければなりません。

例

IBM 拡張

CHAR (88) は値 'X' を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
CHAR	任意の整数	デフォルト文字	可 1

注:

1. IBM 拡張: デフォルト以外の整数の引き数を指定するための機能。
2. XL Fortran は、ASCII 照合順序のみをサポートしています。

IBM 拡張 の終り

CMPLX (X, Y, KIND)

複素数タイプに変換します。

引き数タイプおよび属性

X タイプは整数、実数、複素数のいずれかでなければなりません。

Y (オプション) タイプは整数または実数でなければなりません。 X のタイプが複素数の場合には、Y は指定できません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- タイプは複素数です。
- KIND が存在する場合は、 kind 型付きパラメーターは KIND で指定されたタイプになり、それ以外の場合は、デフォルトの実数タイプになります。

結果値

- Y が存在せず、X が複素数でない場合は、Y がゼロという値をもって存在しているかのようになります。
- Y が存在せず、X が複素数である場合は、Y が値 AIMAG(X) を持って存在しているかのようになります。
- CMPLX(X, Y, KIND) は、実数部分が REAL(X, KIND) で、虚数部分が REAL(Y, KIND) である複素数を持ちます。

例

CMPLX (-3) は値 (-3.0, 0.0) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
CMPLX **1**	デフォルトの実数	デフォルトの複素数	なし

注:

1. IBM 拡張

関連情報

579 ページの『DCMPLX (X, Y)』、 650 ページの『QCMPLX (X, Y)』を参照してください。

CONJG (Z)

共役複素数を求めます。

引き数タイプおよび属性

Z タイプは複素数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

Z と同じです。

結果値

Z が値 (x, y) を持っている場合は、結果は値 (x, -y) を持ちます。

例

CONJG ((2.0, 3.0)) は値 (2.0, -3.0) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
CONJG	デフォルトの複素数	デフォルトの複素数	あり
DCONJG **1**	倍精度複素数	倍精度複素数	あり
QCONJG **1**	COMPLEX(16)	COMPLEX(16)	あり

注:

1. IBM 拡張

COS (X)

コサイン (余弦) 関数です。

引き数タイプおよび属性

X タイプは実数または複素数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- cos(X) とほぼ同じ値になります。
- X のタイプが実数の場合は、X はラジアンと見なされます。
- X のタイプが複素数の場合は、X の実数部分と虚数部分がラジアンの値と見なされます。

例

COS (1.0) の値は 0.54030231 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
COS	デフォルトの実数	デフォルトの実数	あり
DCOS	倍精度実数	倍精度実数	あり
QCOS **1**	REAL(16)	REAL(16)	あり
CCOS **2a**	デフォルトの複素数	デフォルトの複素数	あり
CDCOS **1** **2b**	倍精度複素数	倍精度複素数	あり
ZCOS **1** **2b**	倍精度複素数	倍精度複素数	あり
CQCOS **1** **2b**	COMPLEX(16)	COMPLEX(16)	あり

注:

1. IBM 拡張
2. X が a + bi という形式の複素数であるとすると、以下のようになります (ただし、i = $(-1)^{½}$)。
   a. abs(b) は 88.7228 以下でなければなりません (a は任意の実数値)。
   b. abs(b) は 709.7827 以下でなければなりません (a は任意の実数値)。

COSD (X)

IBM 拡張

コサイン (余弦) 関数です。引き数は「度」の単位となります。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- cos(X) とほぼ同じ値になります。この X 「度」の単位の値を持ちます。

例

COSD (45.0) は値 0.7071067691 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
COSD	デフォルトの実数	デフォルトの実数	あり
DCOSD	倍精度実数	倍精度実数	あり
QCOSD	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

COSH (X)

双曲線コサイン (双曲線余弦) 関数です。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は、cosh(X) の近似値になります。

例

COSH (1.0) は値 1.5430806 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
COSH **1a**	デフォルトの実数	デフォルトの実数	あり
DCOSH **1b**	倍精度実数	倍精度実数	あり
QCOSH **1b** **2**	REAL(16)	REAL(16)	あり

注:

1. X が a + bi という形式の複素数であるとすると、以下のようになります (ただし、i = $(-1)^{½}$)。
   a. abs(b) は 88.7228 以下でなければなりません (a は任意の実数値)。
   b. abs(b) は 709.7827 以下でなければなりません (a は任意の実数値)。
2. IBM 拡張

COUNT (MASK, DIM)

論理配列全体内、または単一次元に沿った個々のベクトル内の真の配列エレメントの数をカウントします。通常、論理配列は、別の組み込み配列でマスクとして使用される配列です。

引き数タイプおよび属性

MASK
論理配列です。

DIM (オプション)
1 ≤ DIM ≤ rank(MASK) の範囲内の整数スカラーです。対応する実引き数は、オプションの仮引き数であってはなりません。

クラス

変換関数

結果値

DIM が存在する場合は、結果はランク rank(MASK)-1 の整数配列になります。 DIM が脱落している場合、または MASK のランクが 1 の場合は、結果はスカラーになります。

その結果作成された配列の各エレメント (R(s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n)) は、対応する次元 (s_1, s_2, ..., $s_{(DIM-1)}$, :, $s_{(DIM+1)}$, ..., s_n) に沿った MASK 内の真のエレメントの数に等しくなります。

MASK がゼロにサイズ決定されている配列であるとすると、結果はゼロに等しくなります。

例

```
! A is the array | T F F |, and B is the array | F F T |
!                | F T T |                     | T T T |

! How many corresponding elements in A and B
! are equivalent?
    RES = COUNT(A .EQV. B)       ! result RES is 3

! How many corresponding elements are equivalent
! in each column?
    RES = COUNT(A .EQV. B, DIM=1) ! result RES is (0,2,1)

! Same question, but for each row.
    RES = COUNT(A .EQV. B, DIM=2) ! result RES is (1,2)
```

CPU_TIME(TIME)

Fortran 95

すべてのスレッドの現行プロセス、および、場合によっては子プロセスにかかる CPU 時間を秒単位で戻します。 **CPU_TIME** を呼び出すと、プログラムの始まりからの処理にかかるプロセッサー時間を戻します。計測される時間は、プログラムが実際に実行された時間であり、プログラムが中断している時間または待っている時間は含まれません。

引き数タイプおよび属性

TIME 実数タイプのスカラーです。これは、**INTENT(OUT)** 引き数で、プロセッサー時間の近似値を割り当てられます。この時間は秒単位で計測されます。 **CPU_TIME** によって戻される時間は、**XLFRTEOPTS** 環境変数実行時オプション **cpu_time_type** の設定によって異なります。 **cpu_time_type** の有効な設定値は次のとおりです。

usertime 現行プロセスのユーザー時間です。ユーザー時間の

定義については、「*AIX Performance and Tuning Guide*」を参照してください。

systime
: 現行プロセスのシステム時間です。システム時間の定義については、「*AIX Performance and Tuning Guide*」を参照してください。

alltime
: 現行プロセスのシステム時間とユーザー時間の合計です。

total_usertime
: 現行プロセスのユーザー時間の合計です。ユーザー時間の合計とは、現行プロセスのユーザー時間の合計と、その子プロセス (ある場合) のユーザー時間の合計です。

total_systime
: 現行プロセスのシステム時間の合計です。システム時間の合計とは、現行プロセスのシステム時間の合計と、その子プロセス (ある場合) のシステム時間の合計です。

total_alltime
: 現行プロセスのユーザー時間とシステム時間の合計です。ユーザー時間とシステム時間の合計とは、現行プロセスのユーザー時間とシステム時間の合計と、その子プロセス (存在する場合) のユーザー時間とシステム時間の合計です。

これが、**cpu_time_type** 実行時オプションを設定していない場合の、**CPU_TIME** の時間のデフォルト測定です。

cpu_time_type 実行時オプションの設定は、**setrteopts** プロシージャーを使用して行います。 **cpu_time_type** 設定へのそれぞれの変更は、後続の **CPU_TIME** の呼び出しすべてに影響します。

クラス

サブルーチン

例

例 1:

```
! The default value for cpu_time_type is used
REAL T1, T2
...         ! First chunk of code to be timed
CALL CPU_TIME(T1)
...         ! Second chunk of code to be timed
CALL CPU_TIME(T2)
```

```
print *, 'Time taken for first chunk of code: ', T1, 'seconds.'
print *, 'Time taken for both chunks of code: ', T2, 'seconds.'
print *, 'Time for second chunk of code was ', T2-T1, 'seconds.'
```

cpu_time_type 実行時オプションを **usertime** に設定したい場合、ksh または bsh コマンド行から次のコマンドを入力します。

```
export XLFRTEOPTS=cpu_time_type=usertime
```

例 2:

```
! Use setrteopts to set the cpu_time_type run-time option as many times
! as you need to
CALL setrteopts ('cpu_time_type=alltime')
CALL stallingloop
CALL CPU_TIME(T1)
print *, 'The sum of the user and system time is', T1, 'seconds'.
CALL setrteopts ('cpu_time_type=usertime')
CALL stallingloop
CALL CPU_TIME(T2)
print *, 'The total user time from the start of the program is', T2, 'seconds'.
```

関連情報

- 詳細については、「ユーザーズ・ガイド」の **XLFRTEOPTS** 環境変数の説明を参照してください。
- 詳細については、 899 ページのsetrteoptsと「ユーザーズ・ガイド」の 『″setrteopts″ サービスおよびユーティリティー・プロシージャー』の説明を参照してください。

Fortran 95 の終り

CSHIFT (ARRAY, SHIFT, DIM)

配列の所定の次元に沿ったすべてのベクトルのエレメントをシフトします。シフトは循環です。つまり、一方の端がシフトオフされたエレメントは、もう一方の端に再び挿入されます。

引き数タイプおよび属性

ARRAY
任意のタイプの配列です。

SHIFT
ARRAY のランクが 1 の場合はスカラー整数でなければなりません。ARRAY のランクが 1 でない場合は、ランク rank(ARRAY)-1 のスカラー整数または整数式になります。

DIM (オプション)
1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。指定しない場合、デフォルトは 1 になります。

クラス

変換関数

結果値

結果は、ARRAY と同じ形状とデータ型を持つ配列になります。

SHIFT がスカラーの場合は、個々のベクトルに同じシフトが適用されます。それ以外の場合は、個々のベクトル ARRAY (s_1, s_2, ..., $s_{(DIM-1)}$, :, $s_{(DIM+1)}$, ..., s_n) は、SHIFT (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) 内の対応する値に従ってシフトされます。

SHIFT の絶対値は、シフトの量を決定します。 SHIFT の符号は、シフトの方向を決定します。

正の SHIFT
: ベクトルの個々のエレメントをベクトルの先頭に向かって移動します。

負の SHIFT
: ベクトルの個々のエレメントをベクトルの終わりに向かって移動します。

ゼロ SHIFT
: シフトを行いません。ベクトルの値は変更されません。

例

```
! A is the array | A D G |
!                | B E H |
!                | C F I |

! Shift the first column down one, the second column
! up one, and leave the third column unchanged.
       RES = CSHIFT (A, SHIFT = (/-1,1,0/), DIM = 1)
! The result is | C E G |
!               | A F H |
!               | B D I |

! Do the same shifts as before, but on the rows
! instead of the columns.
       RES = CSHIFT (A, SHIFT = (/-1,1,0/), DIM = 2)
! The result is | G A D |
!               | E H B |
!               | C F I |
```

CVMGx (TSOURCE, FSOURCE, MASK)

IBM 拡張

条件付きベクトル・マージ関数 (**CVMGM**、 **CVMGN**、**CVMGP**、**CVMGT**、**CVMGZ**) を使用すると、これらの関数を含んでいる既存のコードを移植することができます。

これらの呼び出しは、以下の関数を呼び出すのに非常に似ています。

```
MERGE ( TSOURCE, FSOURCE, arith_expr .op. 0 )
または
MERGE ( TSOURCE, FSOURCE, logical_expr .op. .TRUE. )
```

MERGE 組み込み関数は Fortran 90 の一部なので、新しいプログラムには、これらの関数の代わりに、この組み込み関数を使用することをお勧めします。

引き数タイプおよび属性

TSOURCE
: **LOGICAL**、**INTEGER**、または **REAL** タイプ (1 を除く) の任意の種類のスカラー式または配列式です。

FSOURCE
: TSOURCE と同じタイプおよび型付きパラメーターを持つスカラー式または配列式です。

MASK
: **INTEGER** または **REAL** タイプ (**CVMGM**、**CVMGN**、**CVMGP**、**CVMGZ** の場合) あるいは **LOGICAL** タイプ (**CVMGT** の場合) の任意の種類 (1 を除く) のスカラー式または整数式です。それが配列である場合には、形状が TSOURCE および FSOURCE に従っていなければなりません。

TSOURCE と FSOURCE のどちらか一方だけがタイプなしの場合は、タイプなしの引き数が他方の引き数のタイプを獲得します。 TSOURCE と FSOURCE のどちらの引き数もタイプなしの場合は、両方の引き数が MASK のタイプを獲得します。 MASK もタイプなしの場合は、TSOURCE と FSOURCE の両方がデフォルト整数であると見なされます。 MASK がタイプなしの場合は、**CVMGT** 関数のデフォルト論理値、およびその他の **CVMGx** 関数のデフォルト整数と見なされます。

クラス

エレメント型関数

結果タイプおよび属性

TSOURCE および FSOURCE と同じです。

結果値

結果の関数は、最初の引き数または 2 番目の引き数の値になります。どちらになるかは、3 番目の引き数に対して実行されるテストの結果によって決まります。引き数が配列の場合は、MASK 配列の個々のエレメントに対して実行され、結果には TSOURCE からのエレメントと、FSOURCE からのエレメントが含まれる場合があります。

表 16. CVMGx 組み込みプロシージャーの結果の値

説明	関数戻り値	総称名
正かゼロかのテスト	MASK≥0 の場合には TSOURCE、MASK<0 の場合には FSOURCE	CVMGP

表 16. *CVMGx* 組み込みプロシージャーの結果の値 (続き)

説明	関数戻り値	総称名
負のテスト	MASK<0 の場合には TSOURCE、 MASK≥0 の場合には FSOURCE	CVMGM
ゼロのテスト	MASK=0 の場合には TSOURCE、 MASK≠0 の場合には FSOURCE	CVMGZ
ゼロ以外のテスト	MASK≠0 の場合には TSOURCE、 MASK=0 の場合には FSOURCE	CVMGN
真のテスト	MASK= .TRUE. の場合には TSOURCE、 MASK=.FALSE. の場合には FSOURCE	CVMGT

IBM 拡張 の終り

DATE_AND_TIME (DATE, TIME, ZONE, VALUES)

リアルタイム・クロックからのデータおよび日付を、ISO 8601:1988 に定義されている表記法と互換性のある形式で戻します。

引き数タイプおよび属性

DATE (オプション)

スカラーで、タイプはデフォルト文字でなければならず、完全な値を含むためには、最低でも長さが 8 でなければなりません。これは、INTENT(OUT) 引き数です。 その左端の 8 文字は CCYYMMDD 形式に設定され、この CC は世紀、YY は年、 MM は月、DD は日を表します。日付が入力されないと、これらの文字は、ブランクに設定されます。

TIME (オプション)

スカラーで、タイプはデフォルト文字でなければならず、完全な値を含むためには、最低でも長さが 10 でなければなりません。これは、INTENT(OUT) 引き数です。 この左端の 10 文字は hhmmss.sss 形式の値に設定され、この hh は時、mm は分、ss.sss は秒とミリ秒です。使用可能なクロックがない場合には、ブランクに設定されます。

ZONE (オプション)

スカラーで、タイプはデフォルト文字でなければならず、完全な値を含むためには、最低でも長さが 5 なければなりません。これは、INTENT(OUT) 引き数です。 この左端の 5 文字は ±hhmm 形式の値に設定され、この hh と mm は 協定世界時 (UTC) に対する時差であり、 hh は時間、mm は分を表します。 使用可能なクロックがない場合には、ブランクに設定されます。

IBM 拡張

smit chtz ファースト・パスによってマシンをセットアップしなかった場合、または **smit** で構成できない時間帯にいる場合には、 **ZONE** の値が正しくないことがあります。 手作業で **TZ** 環境変数を設定するか、または **chtz** コマンドを使用して、時間帯を確実に正しくセットアップすることができます。 **TZ** 変数の形式は、*AIX Files Reference* の **/etc/environment** ファイルの項に記載されています。

IBM 拡張 の終り

VALUES (オプション)

タイプはデフォルト整数で、ランクは 1 でなければなりません。これは、INTENT(OUT) 引き数です。 そのサイズは最低でも 8 なければなりません。 VALUES で戻される値は次のとおりです。

VALUES(1)
年 (たとえば 1998) で、有効な日付がない場合は、-HUGE (0) です。

VALUES(2)
月で、有効な日付がない場合は -HUGE (0) です。

VALUES(3)
日付で、有効な日付がない場合は -HUGE (0) です。

VALUES(4)
協定世界時 (UTC) に関する時間差 (分単位) で、この情報が有効でない場合は -HUGE (0) です。

VALUES(5)
時間で、範囲は 0 から 23 で、クロックが存在しない場合は -HUGE (0) です。

VALUES(6)
分で、範囲は 0 から 59 で、クロックが存在しない場合は -HUGE (0) です。

VALUES(7)
秒で、範囲は 0 から 60 で、クロックが存在しない場合は -HUGE (0) です。

VALUES (8)
ミリ秒で、範囲は 0 から 999 で、クロックが存在しない場合は -HUGE (0) です。

クラス

サブルーチン

例

以下にプログラムを示します。

```
INTEGER DATE_TIME (8)
CHARACTER (LEN = 10) BIG_BEN (3)
CALL DATE_AND_TIME (BIG_BEN (1), BIG_BEN (2), &
                    BIG_BEN (3), DATE_TIME)
```

スイスのジュネーブで 1985 年 4 月 12 日の 15:27:35.5 に実行される場合には、このプログラムは値 19850412 を BIG_BEN(1) に、値 152735.500 を BIG_BEN(2) に、値 +0100 を BIG_BEN(3) に、値 1985, 4, 12, 60, 15, 27, 35, 500 を DATE_TIME にそれぞれ割り当てます。

UTC が CCIR (国際無線通信諮問機関) の勧告 460-2 (グリニッジ標準時とも言う) で規定されていることに注意してください。

DBLE (A)

倍精度実数タイプに変換します。

引き数タイプおよび属性

A タイプは整数、実数、複素数のいずれかでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

倍精度実数

結果値

- A のタイプが倍精度実数の場合は、DBLE(A) = A です。
- A のタイプが整数または実数の場合は、結果は倍精度実数既知数が含むことのできるのと同じ精度の A の有効部分を持ちます。
- A のタイプが複素数の場合は、結果は倍精度実数既知数が含むことのできるのと同じ精度の A の有効部分を持ちます。

例

DBLE (-3) は値 -3.0D0 を持ちます。

IBM 拡張

特定名	引き数タイプ	結果タイプ	引き数渡し
DFLOAT	任意の整数	倍精度実数	なし
DBLE	デフォルトの実数	倍精度実数	なし
DBLEQ	REAL(16)	REAL(8)	なし

IBM 拡張 の終り

DCMPLX (X, Y)

IBM 拡張

倍精度複素数タイプに変換します。

引き数タイプおよび属性

X タイプは整数、実数、複素数のいずれかでなければなりません。

Y (オプション) タイプは整数または実数でなければなりません。 X のタイプが複素数の場合には、Y は指定できません。

クラス

エレメント型関数

結果タイプおよび属性

タイプは倍精度複素数です。

結果値

- Y が存在せず X が複素数でない場合は、Y がゼロという値を持って存在しているかのようになります。
- Y が存在せず、X が複素数である場合は、Y が値 AIMAG(X) を持って存在しているかのようになります。
- DCMPLX(X, Y) は、実数部分が REAL(X, KIND=8) で、虚数部分が REAL(Y, KIND=8) である複素数を持ちます。

例

DCMPLX (-3) は値 (-3.0D0, 0.0D0) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
DCMPLX	倍精度実数	倍精度複素数	なし

関連情報

566 ページの『CMPLX (X, Y, KIND)』、 650 ページの『QCMPLX (X, Y)』を参照してください。

IBM 拡張 の終り

DIGITS (X)

タイプおよび kind 型付きパラメーターが引き数と同じである数字の有効桁数を戻します。

引き数タイプおよび属性

X タイプは整数または実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

IBM 拡張

- X のタイプが整数の場合は、X の有効数字の桁数は次のようになります。

```
    type                    bits
-----------               ------
 integer(1)                  07
 integer(2)                  15
 integer(4)                  31
 integer(8)                  63
```

- X のタイプが実数の場合は、X の有効ビット数は次のようになります。

```
   type                    bits
----------               ------
real(4)                     24
real(8)                     53
real(16)                   106
```

IBM 拡張 の終り

例

IBM 拡張

DIGITS (X) = 63。この X のタイプは integer(8) です。 (542 ページの『データ表示モデル』を参照してください。)

IBM 拡張 の終り

DIM (X, Y)

X-Y の差で、正の場合はその値、それ以外の場合はゼロです。

引き数タイプおよび属性

X タイプは整数または実数でなければなりません。

Y タイプおよび kind 型付きパラメーターは X と同じでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- X > Y の場合は、結果の値は X - Y になります。
- X ≤ Y の場合は、結果の値はゼロになります。

例

DIM (-3.0, 2.0) は値 0.0 を持ちます。 DIM (-3.0, -4.0) は値 1.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
IDIM	任意の整数 1	引き数と同じ	あり
DIM	デフォルトの実数	デフォルトの実数	あり
DDIM	倍精度実数	倍精度実数	あり
QDIM 2	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張: デフォルト以外の整数の引き数を指定するための機能。
2. IBM 拡張

DOT_PRODUCT (VECTOR_A, VECTOR_B)

2 つのベクトルについての内積を算出します。

引き数タイプおよび属性

VECTOR_A 数値データ型または論理データ型を持つベクトルです。

VECTOR_B VECTOR_A が数値タイプの場合は数値タイプ、 VECTOR_A が論理タイプの場合は論理タイプでなければなりません。 VECTOR_A と同じサイズでなければなりません。

クラス

変換関数

結果値

結果は、データ型が 115 ページの表 3 および 122 ページの表 4 に記載されている規則に従って、2 つのベクトルのデータ型によって決まるスカラーになります。

ベクトルがゼロにサイズ決定されている配列の場合は、数値データ型を持っていれば結果はゼロに等しくなり、論理タイプの場合にはゼロになります。

VECTOR_A のタイプが整数または実数の場合は、結果の値は SUM(VECTOR_A * VECTOR_B) に等しくなります。

VECTOR_A のタイプが複素数の場合は、結果は SUM(CONJG(VECTOR_A) * VECTOR_A) に等しくなります。

VECTOR_A が論理タイプの場合は、結果は ANY(VECTOR_A .AND. VECTOR_B) に等しくなります。

例

```
! A is (/ 3, 1, -5 /), and B is (/ 6, 2, 7 /).
      RES = DOT_PRODUCT (A, B)
! calculated as
!   ( (3*6) + (1*2) + (-5*7) )
! = (   18  +    2  +  (-35) )
! =  -15
```

DPROD (X, Y)

倍精度実数積

引き数タイプおよび属性

X タイプはデフォルトの実数でなければなりません。

Y タイプはデフォルトの実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

倍精度実数

結果値

結果は、X と Y の積に等しい値を持ちます。

例

DPROD (-3.0, 2.0) は値 -6.0D0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
DPROD	デフォルトの実数	倍精度実数	あり
QPROD **1**	倍精度実数	REAL(16)	あり

注:

1. IBM 拡張

EOSHIFT (ARRAY, SHIFT, BOUNDARY, DIM)

配列の所定の次元に沿ったすべてのベクトルのエレメントをシフトします。シフトは end-off です。つまり、一方の端をシフトオフされたエレメントは失われ、境界エレメントのコピーはもう一方の端でシフトインされます。

引き数タイプおよび属性

ARRAY 任意のタイプの配列です。

SHIFT ランク 1 を持つ ARRAY の場合は、整数タイプのスカラーです。そうでない場合は、スカラー整数またはランク rank(ARRAY)-1 の整数式です。

BOUNDARY (オプション) ARRAY と同じタイプおよび型付きパラメーターを持ちます。 ARRAY が、ランク 1 である場合、BOUNDARY はスカラーでなければなりません。そうでない場合、これは、スカラーであるか、ランク rank(ARRAY)-1 の式です。

DIM (オプション) 1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。

クラス

変換関数

結果値

結果は ARRAY と同じ形状とデータ型を持つ配列になります。

SHIFT の絶対値は、シフトの量を決定します。 SHIFT の符号は、シフトの方向を決定します。

正の SHIFT
ベクトルの個々のエレメントをベクトルの先頭に向かって移動します。エレメントがベクトルの先頭を過ぎた場合は、その値の代わりに、ベクトルの終わりにある BOUNDARY の対応する値が入ります。

負の SHIFT
ベクトルの個々のエレメントをベクトルの終わりに向かって移動します。エレメントがベクトルの終わりを過ぎた場合は、その値の代わりに、ベクトルの初めにある BOUNDARY の対応する値が入ります。

ゼロ SHIFT
シフトを行いません。ベクトルの値は変更されません。

結果値

BOUNDARY がスカラー値である場合は、この値はすべてのシフト値で使用されます。

BOUNDARY が値の配列である場合は、添え字を持つ (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) を持つ BOUNDARY の配列エレメントの値がその次元について適用されます。

BOUNDARY が指定されないと、以下のデフォルト値が使用されます。どれが使用されるかは、ARRAY のデータ型によって決まります。

文字
'ƀ' (ブランク 1 つ)

論理
false

整数
0

実数
0.0

複素数
(0.0, 0.0)

例

```
! A is | 1.1 4.4 7.7 |, SHIFT is S=(/0, -1, 1/),
!      | 2.2 5.5 8.8 |
!      | 3.3 6.6 9.9 |
! and BOUNDARY is the array B=(/-0.1, -0.2, -0.3/).
```

```
! Leave the first column alone, shift the second
! column down one, and shift the third column up one.
RES = EOSHIFT (A, SHIFT = S, BOUNDARY = B, DIM = 1)
! The result is | 1.1 -0.2  8.8 |
!               | 2.2  4.4  9.9 |
!               | 3.3  5.5 -0.3 |

! Do the same shifts as before, but on the
! rows instead of the columns.
RES = EOSHIFT (A, SHIFT = S, BOUNDARY = B, DIM = 2)
! The result is |  1.1  4.4  7.7 |
!               | -0.2  2.2  5.5 |
!               |  6.6  9.9 -0.3 |
```

EPSILON (X)

引き数と同じタイプおよび kind 型付きパラメーターの数を示すモデル内の単位と比較すると、無視してもよいほど小さい正のモデル番号を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

X と同じタイプおよび kind 型付きパラメーターのスカラーです。

結果値

結果は次のようになります。

$2.0ei0^{1 - \mathrm{DIGITS(X)}}$

上記の *ei* は指数標識 (E、D、または Q のいずれか) で、X のタイプによって決まります。

IBM 拡張

```
type                      EPSILON(X)
----                   ------------------
real(4)                   02E0 ** (-23)
real(8)                   02D0 ** (-52)
real(16)                  02Q0 ** (-105)
```

IBM 拡張 の終り

例

IBM 拡張

X のタイプが real(4) の場合は、EPSILON (X) = 1.1920929E-07 になります。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

ERF (X)

IBM 拡張

誤差関数

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 結果は、erf(X) の近似値になります。
- 結果は、-1 ≤ ERF(X) ≤ 1 の範囲内にあります。

例

ERF (1.0) は値 0.8427007794 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ERF	デフォルトの実数	デフォルトの実数	あり

特定名	引き数タイプ	結果タイプ	引き数渡し
DERF	倍精度実数	倍精度実数	あり
QERF	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

ERFC (X)

IBM 拡張

基本誤差関数

$$\mathrm{erfc}(x)=1-\mathrm{erf}(x)=\frac{2}{\sqrt{\pi}}\int_x^{\infty}e^{-t^2}dt$$

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 結果は 1-ERF(X) に等しい値を持ちます。
- 結果は 0 ≤ ERFC(X) ≤ 2 の範囲内にあります。

例

ERFC (1.0) は値 0.1572992057 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ERFC	デフォルトの実数	デフォルトの実数	あり
DERFC	倍精度実数	倍精度実数	あり

特定名	引き数タイプ	結果タイプ	引き数渡し
QERFC	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

EXP (X)

指数を求めます。

引き数タイプおよび属性

X タイプは実数または複素数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- 結果は e^x に近似した値を持ちます。
- X のタイプが複素数の場合は、その実数部分と虚数部分がラジアンの値と見なされます。

例

EXP (1.0) は値 2.7182818 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
EXP **1**	デフォルトの実数	デフォルトの実数	あり
DEXP **2**	倍精度実数	倍精度実数	あり
QEXP **2** **3**	REAL(16)	REAL(16)	あり
CEXP **4a**	デフォルトの複素数	デフォルトの複素数	あり
CDEXP **4b** **3**	倍精度複素数	倍精度複素数	あり
ZEXP **4b** **3**	倍精度複素数	倍精度複素数	あり
CQEXP **4b** **3**	COMPLEX(16)	COMPLEX(16)	あり

注:

1. X は 88.7228 以下でなければなりません。
2. X は 709.7827 以下でなければなりません。
3. IBM 拡張
4. X が a + bi という形式の複素数であるとき、以下のようになります (ただし、$i = (-1)^{1/2}$)。
   a. a は 88.7228 以下でなければなりません (b は任意の実数値)。
   b. a は 709.7827 以下でなければなりません (b は任意の実数値)。

EXPONENT (X)

モデル番号として表現される場合は、引き数の指数部分を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

- X ≠ 0 の場合、結果は X の指数になります。 (常にデフォルトの指数の範囲内にあります)。
- X = 0 の場合は、X の指数はゼロです。

例

IBM 拡張

EXPONENT (10.2) = 4。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

FLOOR(A, KIND)

その引き数以下の最大整数を戻します。

引き数タイプおよび属性

A タイプは実数でなければなりません。

Fortran 95

KIND (オプション)
スカラー整数の初期化式でなければなりません。

Fortran 95 の終り

クラス

エレメント型関数

結果タイプおよび属性

- タイプは整数です。

Fortran 95

- KIND が存在する場合は、kind 型付きパラメーターは KIND で指定されたものになります。存在しない場合は、KIND 型付きパラメーターはデフォルトの整数タイプになります。

Fortran 95 の終り

結果値

結果は、A 以上の最小整数に等しい値を持ちます。

Fortran 95

指定された KIND の整数として結果を表現できない場合は、この結果は未定義になります。

Fortran 95 の終り

例

```
FLOOR(-3.7) は値 -4 を持ちます。
FLOOR(3.7) は値 3 を持ちます。
```

Fortran 95

FLOOR(1000.1, KIND=2) は値 1 000 と kind 型付きパラメーター 2 を持ちます。

Fortran 95 の終り

FRACTION (X)

引き数値のモデル表現の小数部分を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

IBM 拡張

結果は次のようになります。

X * (2.0$^{-EXPONENT(X)}$)

IBM 拡張 の終り

例

IBM 拡張

FRACTION(10.2) =2^{-4} * 10.2 ≈ 0.6375

IBM 拡張 の終り

GAMMA(X)

IBM 拡張

ガンマ関数

$$\Gamma(x) = \int_0^{\infty} u^{x-1} e^{-u} du$$

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は、Γ(X) の近似値になります。

例

GAMMA (1.0) は値 1.0 を持ちます。

GAMMA (10.0) は値 362880.0 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
GAMMA 1	デフォルトの実数	デフォルトの実数	あり
DGAMMA 2	倍精度実数	倍精度実数	あり
QGAMMA 3	REAL(16)	REAL(16)	あり

X は次の不等式を満たさなければなりません:

1. -2.0**23 < X ≤ 35.0401 (正でない整数値を除く)
2. -2.0**52 < X ≤ 171.6243 (正でない整数値を除く)
3. -2.0**105 < X ≤ 171.6243 (正でない整数値を除く)

IBM 拡張 の終り

GETENV (NAME, VALUE)

IBM 拡張

指定された環境変数の値を判別します。

引き数タイプおよび属性

NAME
オペレーティング・システムの環境変数の名前を識別する文字ストリングです。このストリングは大文字小文字の区別をします。これは、スカラーで、タイプがデフォルトの文字でなければならない INTENT(IN) 引き数です。

VALUE
サブルーチンが戻る前に、環境変数の値を保持します。これは、スカラーで、タイプがデフォルトの文字でなければならない INTENT(OUT) 引き数です。

クラス

サブルーチン

結果値

結果は、関数結果変数としてではなく、VALUE 引き数で戻されます。

NAME 引き数に指定されている環境変数が存在しない場合は、VALUE 引き数にブランクが入ります。

例

```
      CHARACTER (LEN=16)   ENVDATA
      CALL GETENV('HOME', VALUE=ENVDATA)
! Print the value.
      PRINT *, ENVDATA
! Show how it is blank-padded on the right.
      WRITE(*, '(Z32)') ENVDATA
      END
```

上記のプログラムで生成される出力例は次のとおりです。

```
 /home/mark
2F686F6D652F6D61726B202020202020
```

関連情報

オペレーティング・システム・レベルのインプリメンテーションに関する詳細は、「*AIX Technical Reference: Base Operating System and Extensions Volume 1*」の『**getenv** subroutine』を参照してください。

IBM 拡張 の終り

HFIX (A)

IBM 拡張

REAL(4) から **INTEGER(2)** に変換します。

このプロシージャーは特別な関数で、一般的な関数ではありません。

引き数タイプおよび属性

A タイプは **REAL(4)** でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

INTEGER(2) スカラーまたは配列

結果値

- $|A| < 1$ の場合は、INT (A) は値 0 を持ちます。
- $|A| \geq 1$ の場合は、INT (A) は絶対値が A の絶対値を超えない最大整数で、符号が A の符号と同じ整数になります。
- 結果を **INTEGER(2)** で表現できない場合は、結果は不定になります。

例

HFIX (-3.7) は値 -3 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
HFIX	REAL(4)	INTEGER(2)	なし

IBM 拡張 の終り

HUGE (X)

引き数と同じ多数のタイプおよび kind 型付きパラメーターを表すモデル内の最大数を戻します。

引き数タイプおよび属性

X タイプは整数または実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

X と同じタイプおよび kind 型付きパラメーターのスカラーです。

結果値

- X が任意の整数タイプである場合は、結果は次のようになります。

  $2^{\mathrm{DIGITS(X)}} - 1$

- X が任意の実数タイプである場合は、結果は次のようになります。

  $(1.0 - 2.0^{-\mathrm{DIGITS(X)}}) * (2.0^{\mathrm{MAXEXPONENT(X)}})$

例

IBM 拡張

X のタイプが real(8) の場合は、 HUGE (X) = (1D0 - 2D0**-53) * (2D0**1024)。

X のタイプが integer(8) の場合は、HUGE (X) = (2**63) - 1。

542 ページの『データ表示モデル』を参照してください。

IBM 拡張 の終り

IACHAR (C)

ASCII 照合順序内の文字の位置を戻します。

引き数タイプおよび属性

C タイプはデフォルトの文字で、長さは 1 でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

- C が ISO 646:1983 (International Reference Version) に指定されているコードで定義されている照合順序内にある場合は、結果は、その順序内の C の位置になり、以下の不等式 (0 ≤ IACHAR (C) ≤ 127) を満たします。 C が ASCII 照合順序内にない場合は、不定の値が戻されます。
- 結果は、字句比較関数 LGE、LGT、LLE、LLT と一致します。たとえば、LLE (C, D) が真であれば、IACHAR (C) .LE. IACHAR (D) も真です。

例

IACHAR ('X') は値 88 を持ちます。

IAND (I, J)

論理 AND を実行します。

引き数タイプおよび属性

I
タイプは整数でなければなりません。

J
タイプは、I と同じ kind 型付きパラメーターを持つ整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、以下の表に従って、I と J (ビット単位) を結合することによって得られる値になります。

```
I   J   IAND (I,J)
------------------
1   1       1
1   0       0
0   1       0
0   0       0
```

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

IAND (1, 3) は値 1 を持ちます。 542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
IAND **1**	任意の整数	引き数と同じ	あり
AND **1**	任意の整数	引き数と同じ	あり

注:

1. IBM 拡張

IBCLR (I, POS)

1 ビットをゼロにクリアします。

引き数タイプおよび属性

I タイプは整数でなければなりません。

POS タイプは整数でなければなりません。 これは、負以外で、BIT_SIZE (I) よりも小さくなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、I のビット POS がゼロに設定されること以外は、I のビットのシーケンスの値を持ちます。

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

IBCLR (14, 1) の結果は 12 になります。

V が値 (/1, 2, 3, 4/) を持っている場合は、IBCLR (POS = V, I = 31) の値は (/29, 27, 23, 15/) になります。

542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
IBCLR 1	任意の整数	引き数と同じ	あり

注:

1. IBM 拡張

IBITS (I, POS, LEN)

ビットのシーケンスを抽出します。

引き数タイプおよび属性

I タイプは整数でなければなりません。

POS タイプは整数でなければなりません。 これは負以外でなければならず、POS + LEN は BIT_SIZE (I) 以下でなければなりません。

LEN タイプは整数で、負以外でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、右寄せしたビット POS で始まり、他のすべてのビット・ゼロを持つ I 内の LEN ビットのシーケンスの値を持ちます。

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

IBITS (14, 1, 3) は値 7 を持ちます。 542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
IBITS 1	任意の整数	引き数と同じ	あり

注:

1. IBM 拡張

IBSET (I, POS)

1 ビットを 1 に設定します。

引き数タイプおよび属性

I タイプは整数でなければなりません。

POS タイプは整数でなければなりません。 これは、負以外で、BIT_SIZE (I) よりも小さくなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、I のビット POS が 1 に設定されること以外は、I のビットのシーケンスの値を持ちます。

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

IBSET (12, 1) は値 14 になります。

V が値 (/1, 2, 3, 4/) を持っている場合は、IBSET (POS = V, I = 0) の値は (/2, 4, 8, 16/) になります。

542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
IBSET 1	任意の整数	I と同じです。	あり

注:

1. IBM 拡張

ICHAR (C)

文字の kind 型付きパラメーターと関連した照合文字列内の文字の位置を戻します。

引き数タイプおよび属性

C タイプは文字でなければならず、長さは 1 でなければなりません。その値は、表示可能な文字の値でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

- 結果は、C の kind 型付きパラメーターと関連した照合文字列内の C の位置になり、0 ≤ ICHAR (C) ≤ 127 の範囲内にあります。
- 表示可能文字 C および D の場合は、C .LE. D は ICHAR (C) .LE. ICHAR (D) が真の場合のみ真で、C .EQ. D は ICHAR (C) ·EQ· ICHAR (D) が真の場合のみ真です。

例

IBM 拡張

ICHAR ('X') は ASCII 照合文字列内の値 88 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ICHAR	デフォルト文字	デフォルトの整数	可 **1**

注:

1. この拡張機能は、名前を引き数として渡す機能です。
2. XL Fortran は、ASCII 照合順序のみをサポートしています。

IBM 拡張 の終り

IEOR (I, J)

排他論理和を実行します。

引き数タイプおよび属性

I タイプは整数でなければなりません。

J タイプは、I と同じ KIND 型付きパラメーターを持つ整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、以下の真理値表に従って、I と J (ビット単位) を組み合わせることによって得られる値になります。

```
I   J  IEOR (I,J)
-----------------
1   1      0
1   0      1
0   1      1
0   0      0
```

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

IEOR (1, 3) は値 2 を持ちます。 542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
IEOR **1**	任意の整数	引き数と同じ	あり
XOR **1**	任意の整数	引き数と同じ	あり

注:

1. IBM 拡張

ILEN (I)

IBM 拡張

整数の 2 の補数表現の、ビット単位の長さよりも小さい値を戻します。

引き数タイプおよび属性

I タイプは整数です。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

- I が負の場合は、ILEN(I)=CEILING(LOG2(-I))
- I が負の値でない場合は、ILEN(I)=CEILING(LOG2(I+1))

例

```
I=ILEN(4)  ! 3
J=ILEN(-4) ! 2
```

IBM 拡張 の終り

IMAG (Z)

IBM 拡張

AIMAG と同じです。

関連情報

550 ページの『AIMAG (Z), IMAG (Z)』。

IBM 拡張 の終り

INDEX (STRING, SUBSTRING, BACK)

ストリング内のサブストリングの開始位置を戻します。

引き数タイプおよび属性

STRING タイプは文字でなければなりません。

SUBSTRING タイプは STRING と同じ kind 型付きパラメーターを持つ文字でなければなりません。

BACK (オプション)
タイプは論理タイプでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

- ケース (i): BACK がなかったり、値 .FALSE. を持って存在している場合は、結果は I の最小の正の値になります。たとえば、そのような値が存在しない場合には、STRING (I : I + LEN (SUBSTRING) - 1) = SUBSTRING またはゼロになります。LEN (STRING) < LEN (SUBSTRING) の場合には、ゼロが戻されます。 LEN (SUBSTRING) = 0 の場合には、1 が戻されます。
- ケース (ii): BACK が値 .TRUE. を持って存在している場合は、結果は LEN (STRING) - LEN (SUBSTRING) + 1 以下の I の最大値になります。たとえば、そのような値がない場合には、STRING (I : I + LEN (SUBSTRING) - 1) = SUBSTRING またはゼロになります。 LEN (STRING) < LEN (SUBSTRING) の場合にはゼロが戻され、LEN (STRING) + 1 の場合には LEN (SUBSTRING) = 0 が戻されます。

例

INDEX ('FORTRAN','R') は値 3 を持ちます。

INDEX ('FORTRAN', 'R', BACK = .TRUE.) は値 5 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
INDEX	デフォルト文字	デフォルトの整数	可 **1**

注:

1. この特定名が引き数として渡されると、プロシージャーは任意引き数 **BACK** なしでしか参照できなくなります。

INT (A, KIND)

整数タイプに変換します。

引き数タイプおよび属性

A タイプは整数、実数、複素数のいずれかでなければなりません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 整数
- KIND が存在する場合は、kind 型付きパラメーターは KIND で指定されたものになります。存在しない場合は、kind タイプ・パラメーターはデフォルトの整数タイプになります。

結果値

- ケース (i): A のタイプが整数の場合は、INT (A) = A です。
- ケース (ii): A のタイプが実数の場合は、次の 2 つのケースがあります。 $|A| < 1$ の場合は、INT (A) は値 0 を持ち、$|A| \geq 1$ の場合は、INT (A) は絶対値が A の絶対値を超えない最大整数で、符号が A の符号と同じである整数になります。
- ケース (iii): A のタイプが複素数の場合は、INT (A) は、ケース (ii) の規則を A の実数部分に適用することによって得られる値になります。
- この値を指定された整数タイプで表現できない場合は、結果は不定になります。

例

INT (-3.7) は値 -3 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
INT	デフォルトの実数	デフォルトの整数	なし
IDINT	倍精度実数	デフォルトの整数	なし
IFIX	デフォルトの実数	デフォルトの整数	なし
IQINT 1	REAL(16)	デフォルトの整数	なし

注:

1. IBM 拡張

関連情報

プログラムを XL Fortran に移植したときの **INT** の別の動作については、「ユーザーズ・ガイド」の **-qport** コンパイラー・オプションを参照してください。

INT2(A)

IBM 拡張

実数値または整数値を 2 バイトの整数に変換します。

引き数タイプおよび属性

A 整数または実数タイプのスカラーでなければなりません。

INT2 を別の関数呼び出しの実引き数として渡すことはできません。

クラス

エレメント型関数

結果タイプおよび属性

INTEGER(2) スカラー

結果値

A のタイプが整数の場合は、INT2(A) = A です。

A のタイプが実数の場合は、以下の 2 つの可能性があります。

- |A| < 1 の場合、INT2(A) の値は 0 です。
- |A| >= 1 の場合、INT2(A) は、絶対値が *A* の絶対値を超えない最大整数で、符号が *A* の符号と同じ整数になります。

どちらも場合でも、切り捨てが行われる場合があります。

例

以下は、INT2 関数の例です。

```
REAL*4 :: R4
REAL*8 :: R8
INTEGER*4 :: I4
INTEGER*8 :: I8

R4 = 8.8; R8 = 18.9
I4 = 4; I8 = 8
PRINT *, INT2(R4), INT2(R8), INT2(I4), INT2(I8)
PRINT *, INT2(2.3), INT2(6)
PRINT *, INT2(65535.78), INT2(65536.89)
END
```

上記のプログラムで生成される出力例は次のとおりです。

```
8 18 4 8
2 6
-1 0      ! The results indicate that truncation has occurred, since
          ! only the last two bytes were saved.
```

IBM 拡張 の終り

IOR (I, J)

包含論理和を実行します。

引き数タイプおよび属性

I タイプは整数でなければなりません。

J タイプは、I と同じ kind 型付きパラメーターを持つ整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、以下の真理値表に従って、I と J (ビット単位) を組み合わせることによって得られる値になります。

```
I   J   IOR (I,J)
-----------------
1   1       1
1   0       1
0   1       1
0   0       0
```

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

IOR (1, 3) は値 3 を持ちます。 542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
IOR 1	任意の整数	引き数と同じ	あり
OR 1	任意の整数	引き数と同じ	あり

注:

1. IBM 拡張

ISHFT (I, SHIFT)

論理シフトを実行します。

引き数タイプおよび属性

I
タイプは整数でなければなりません。

SHIFT
タイプは整数でなければなりません。 SHIFT の絶対値は、BIT_SIZE (I) 以下でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

- 結果は、I のビットを SHIFT 位置だけシフトすることによって得られる値になります。
- SHIFT が正の場合はシフトは左へ行われ、SHIFT が負の場合は右へ行われます。そして、SHIFT がゼロの場合は、シフトは行われません。
- 左または右から適切にシフトアウトされたビットは失われます。
- 空のビットはゼロで埋め込まれます。
- ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

ISHFT (3, 1) は結果 6 を持ちます。 542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
ISHFT **1**	任意の整数	引き数と同じ	あり

注:

1. IBM 拡張

ISHFTC (I, SHIFT, SIZE)

右端ビットの循環シフトを実行します。つまり、一方の端を超えてシフトされたビットは、もう一方の端に再び挿入されます。

引き数タイプおよび属性

I
タイプは整数でなければなりません。

SHIFT
タイプは整数でなければなりません。 SHIFT の絶対値は、SIZE 以下でなければなりません。

SIZE (オプション)
タイプは整数でなければなりません。 SIZE の値は正でなければならず、BIT_SIZE (I) を超えてはなりません。 SIZE が存在しない場合は、BIT_SIZE (I) の値を持って存在しているかのようになります。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、I の右端ビット SIZE を SHIFT 位置だけ循環シフトすることによって得られる値になります。 SHIFT が正の場合はシフトは左へ行われ、SHIFT が負の場合は右へ行われます。そして、SHIFT がゼロの場合は、シフトは行われません。ビットはまったく失われません。シフトされなかったビットは変更されません。

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

ISHFTC (3, 2, 3) は値 5 を持ちます。 542 ページの『整数ビット・モデル』を参照してください。

IBM 拡張

特定名	引き数タイプ	結果タイプ	引き数渡し
ISHFTC	任意の整数	引き数と同じ	可 1

注:

1. この特定名が引き数として渡されると、プロシージャーは 3 つの引き数を全部使用しないと参照できません。

IBM 拡張 の終り

KIND (X)

X の kind 型付きパラメーターの値を戻します。

引き数タイプおよび属性

X 任意の組み込みタイプを使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

結果は、X の kind 型付きパラメーターの値に等しい値になります。

XL Fortran がサポートしている kind 型付きパラメーターについては、 29 ページの『組み込み型』を参照してください。

例

KIND (0.0) は、デフォルトの実数タイプの kind 型付きパラメーター値を持ちます。

LBOUND (ARRAY, DIM)

配列内の個々の次元の下限、または、指定された次元の下限を戻します。

引き数タイプおよび属性

ARRAY 下限をユーザーが決定する配列です。その境界を定義しなければなりません。つまり、関連解除したポインター、または、割り振りされていない割り振り可能配列であってはなりません。

DIM (オプション)
$1 \le DIM \le rank(ARRAY)$ の範囲内の整数スカラーです。 対応する実引き数は、オプションの仮引き数であってはなりません。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数

DIM が存在する場合は、結果はスカラーになります。 DIM が存在しない場合は、結果は ARRAY 内の個々の次元に対してエレメントを 1 つ持っている 1 次元の配列になります。

結果値

結果の中の個々のエレメントは、配列の次元に対応します。

- ARRAY が全体配列または配列構造体コンポーネントである場合は、LBOUND(ARRAY, DIM) は、ARRAY の添え字 DIM の下限に等しくなります。

  唯一の例外はゼロにサイズ決定され、ARRAY が DIM ランクの配列に決定されていない次元の場合です。この場合には、下限に対して宣言されている値とは無関係に結果内の対応するエレメントは 1 になります。
- ARRAY が全体配列または配列構造体コンポーネントでない配列セクションまたは式である場合は、個々のエレメントは値 1 を持ちます。

例

```
        REAL A(1:10, -4:5, 4:-5)

        RES=LBOUND( A )
! The result is (/ 1, -4, 1 /).

        RES=LBOUND( A(:,:,:) )
        RES=LBOUND( A(4:10,-4:1,:) )
! The result in both cases is (/ 1, 1, 1 /)
! because the arguments are array sections.
```

LEADZ (I)

IBM 拡張

整数の 2 進表現の中の先行ゼロ・ビットの数を戻します。

引き数タイプおよび属性

I タイプは整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、整数の最左端の 1 ビットより左側にある 0 ビットのカウントです。

例

```
I = LEADZ(0_4)  ! I=32
J = LEADZ(4_4)  ! J=29
K = LEADZ(-1_4) ! K=0
```

IBM 拡張 の終り

LEN (STRING)

文字エンティティーの長さを戻します。この関数の引き数は、定義する必要はありません。

引き数タイプおよび属性

STRING タイプは文字でなければなりません。スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

結果は、スカラーの場合は STRING 内の文字数に等しい値を持ち、配列値の場合には STRING のエレメント内の文字数に等しい値を持ちます。

例

C が以下のステートメントで宣言される場合は、

```
CHARACTER (11) C(100)
```

LEN (C) は値 11 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LEN	デフォルト文字	デフォルトの整数	可 **1**

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LEN_TRIM (STRING)

後続ブランク文字をカウントせずに、文字引き数の長さを戻します。

引き数タイプおよび属性

STRING タイプは文字でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

結果は、STRING 内の後続ブランクの後に残っている文字の数に等しい値になります。引き数の中に非ブランク文字が入っていない場合は、結果はゼロになります。

例

LEN_TRIM ('ƀAƀBƀ') は値 4 を持ちます。 LEN_TRIM ('ƀƀ') は値 0 を持ちます。

LGAMMA (X)

IBM 拡張

ガンマ関数の対数

$$\log_e \Gamma(x) = \log_e \int_0^{\infty} u^{x-1} e^{-u} du$$

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は $\log_e \Gamma(X)$ に等しい値になります。

例

LGAMMA (1.0) は値 0.0 を持ちます。

LGAMMA (10.0) は値 12.80182743 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LGAMMA	デフォルトの実数	デフォルトの実数	あり
LGAMMA	倍精度実数	倍精度実数	あり
ALGAMA 1	デフォルトの実数	デフォルトの実数	あり
DLGAMA 2	倍精度実数	倍精度実数	あり
QLGAMA 3	REAL(16)	REAL(16)	あり

X は次の不等式を満たさなければなりません:

1. $0 < X \leq 4.0850E36$
2. $2.3561D\text{-}304 \leq X \leq 2^{1014}$
3. $2.3561Q\text{-}304 \leq X \leq 2^{1014}$

IBM 拡張 の終り

LGE (STRING_A, STRING_B)

ASCII 照合順序に基づいて、ストリングが別のストリングよりも字句的に大きいか、それとも等しいかをテストします。

引き数タイプおよび属性

STRING_A タイプはデフォルトの文字でなければなりません。

STRING_B タイプはデフォルトの文字でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの論理値

結果値

- ストリングの長さが等しくない場合は、短い方のストリングの右にブランクを入れて、長い方のストリングと同じ長さになるまで拡張したものと想定して、比較が実行されます。
- ASCII 文字セットにない文字がストリングに含まれている場合は、結果は不定になります。
- ストリングが等しい場合、あるいは、ASCII 照合順序で STRING_B の後に STRING_A が続いている場合は結果は真になり、それ以外の場合は偽になります。STRING_A と STRING_B の長さが両方ともゼロの場合は、結果は真になることに注意してください。

例

LGE ('ONE','TWO') は、値 .FALSE. を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LGE	デフォルト文字	デフォルトの論理値	可 1

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LGT (STRING_A, STRING_B)

ASCII 照合順序に基づいて、ストリングが別のストリングよりも字句的に大きいかどうかをテストします。

引き数タイプおよび属性

STRING_A タイプはデフォルトの文字にします。

STRING_B タイプはデフォルトの文字にします。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの論理値

結果値

- ストリングの長さが等しくない場合は、短い方のストリングの右にブランクを入れて、長い方のストリングと同じ長さになるまで拡張したものと想定して、比較が実行されます。

- ASCII 文字セットにない文字がストリングに含まれている場合は、結果は不定になります。
- ASCII 照合順序の STRING_B の後に STRING_A が続いている場合は結果は真になり、それ以外の場合は偽になります。 STRING_A と STRING_B の長さが両方ともゼロの場合、結果は偽になることに注意してください。

例

LGT ('ONE','TWO') は、値 .FALSE. を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LGT	デフォルト文字	デフォルトの論理値	可 1

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LLE (STRING_A, STRING_B)

ASCII 照合順序に基づいて、ストリングが別のストリングよりも字句的に小さいか、それとも等しいかをテストします。

引き数タイプおよび属性

STRING_A タイプはデフォルトの文字にします。

STRING_B タイプはデフォルトの文字にします。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの論理値

結果値

- ストリングの長さが等しくない場合は、短い方のストリングの右にブランクを入れて、長い方のストリングと同じ長さになるまで拡張したものと想定して、比較が実行されます。
- ASCII 文字セットにない文字がストリングに含まれている場合は、結果は不定になります。

- ストリングが等しい場合、または、ASCII 照合順序の STRING_B の前に STRING_A がある場合は、結果は真になり、それ以外の場合は結果は偽になります。 STRING_A と STRING_B の長さが両方ともゼロの場合は、結果は真になることに注意してください。

例

LLE (’ONE’,’TWO’) は、値 .TRUE. を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LLE	デフォルト文字	デフォルトの論理値	可 1

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LLT (STRING_A, STRING_B)

ASCII 照合順序に基づいて、ストリングが別のストリングよりも字句的に小さいかどうかをテストします。

引き数タイプおよび属性

STRING_A　タイプはデフォルトの文字にします。

STRING_B　タイプはデフォルトの文字にします。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの論理値

結果値

- ストリングの長さが等しくない場合は、短い方のストリングの右にブランクを入れて、長い方のストリングと同じ長さになるまで拡張したものと想定して、比較が実行されます。
- ASCII 文字セットにない文字がストリングに含まれている場合は、結果は不定になります。
- ASCII 照合順序の STRING_B の前に STRING_A がある場合、結果は真になり、それ以外の場合は結果は偽になります。 STRING_A と STRING_B の長さが両方ともゼロの場合、結果は偽になることに注意してください。

例

LLT ('ONE','TWO') は、値 .TRUE. を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LLT	デフォルト文字	デフォルトの論理値	可 **1**

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LOC (X)

IBM 拡張

整数 **POINTER** を定義するのに使用できる X のアドレスを戻します。

引き数タイプおよび属性

X ユーザーがアドレスを検出するデータ・オブジェクトです。不定のポインターや関連解除されているポインター、あるいはパラメーターであってはなりません。配列がゼロ・サイズの場合は、ゼロ・サイズ以外のストレージ順序と関連したストレージでなければなりません。配列セクションの場合は、配列セクションのストレージが連続域でなければなりません。

クラス

照会関数

結果タイプおよび属性

結果は、32 ビット・モードのタイプ INTEGER(4)、および 64 ビット・モードのタイプ INTEGER(8) です。

結果値

結果はデータ・オブジェクトのアドレスになり、X がポインターの場合は、関連したターゲットのアドレスになります。引き数が無効であると、結果は不定です。

例

```
INTEGER A,B
POINTER (P,I)

P=LOC(A)
P=LOC(B)
END
```

IBM 拡張 の終り

LOG (X)

自然対数を求めます。

引き数タイプおよび属性

X タイプは実数または複素数でなければなりません。

- X が実数の場合は、値はゼロよりも大きくなければなりません。
- X が複素数の場合は、値はゼロ以外でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- $\log_e X$ に近似する値を持ちます。
- 複素数引き数の場合は、LOG ((a,b)) は LOG (ABS((a,b))) + ATAN2((b,a)) に近似します。

  引き数のタイプが複素数の場合は、結果は $-\pi < \omega \le \pi$ の範囲内の ω のプリンシパル値になります。引き数の実数部分がゼロよりも小さく、虚数部分がゼロの場合には、結果の虚数部分は π に近似します。

例

LOG (10.0) は値 2.3025851 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ALOG	デフォルトの実数	デフォルトの実数	あり
DLOG	倍精度実数	倍精度実数	あり
QLOG	REAL(16)	REAL(16)	可 1
CLOG	デフォルトの複素数	デフォルトの複素数	あり
CDLOG	倍精度複素数	倍精度複素数	可 1
ZLOG	倍精度複素数	倍精度複素数	可 1
CQLOG	COMPLEX(16)	COMPLEX(16)	可 1

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LOG10 (X)

常用対数を求めます。

引き数タイプおよび属性

X タイプは実数でなければなりません。 X の値はゼロよりも大きくなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は $\log_{10}X$ に等しい値になります。

例

LOG10 (10.0) は値 1.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
ALOG10	デフォルトの実数	デフォルトの実数	あり
DLOG10	倍精度実数	倍精度実数	あり
QLOG10	REAL(16)	REAL(16)	あり **1**

注:

1. IBM 拡張: 名前を引き数として渡す機能。

LOGICAL (L, KIND)

異なる kind 型付きパラメーター値を持つ論理タイプのオブジェクト間で変換を行います。

引き数タイプおよび属性

L タイプは論理タイプでなければなりません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 論理タイプ
- KIND が存在する場合は、kind 型付きパラメーターは KIND で指定されたものになります。存在しない場合は、kind 型付きパラメーターはデフォルトの論理タイプになります。

結果値

値は L の値になります。

例

LOGICAL (L .OR. .NOT. L) は値 .TRUE. を持ち、論理変数 L の kind タイプ・パラメーターとは無関係に、タイプはデフォルトの論理タイプになります。

LSHIFT (I, SHIFT)

IBM 拡張

左への論理シフトを実行します。

引き数タイプおよび属性

I
タイプは整数でなければなりません。

SHIFT
タイプは整数でなければなりません。 負以外で、BIT_SIZE(I) 以下でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

- 結果は、I のビットを SHIFT 位置だけ左にシフトすることによって得られる値になります。

- 空のビットはゼロで埋め込まれます。
- ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

LSHIFT (3, 1) は結果 6 を持ちます。

LSHIFT (3, 2) は結果 12 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
LSHIFT	任意の整数	引き数と同じ	あり

IBM 拡張 の終り

MATMUL(MATRIX_A, MATRIX_B, MINDIM)

マトリックス乗算を実行します。

引き数タイプおよび属性

MATRIX_A
ランクが 1 または 2 で、データ型が数値または論理の配列です。

MATRIX_B
ランクが 1 または 2 で、データ型が数値または論理の配列です。これは MATRIX_A とは異なる数値タイプの場合もありますが、 1 つの数値マトリックスに 1 つの論理マトリックスという使い方はできません。

IBM 拡張

MINDIM (オプション)
Strassen アルゴリズムの Winograd 変分 (大きなマトリックスに対しては速い場合がある) を使用してマトリックス乗算を実行するかどうかを決定する整数です。このアルゴリズムは、サブマトリックスのエクステントが MINDIM よりも小さくなるまで、オペランドのマトリックスを 4 つのほぼ等しい部分に再帰的に分割します。

注: Strassen の方式は、入力マトリックスのある一定の行または列の位取りに対して安定ではありません。したがって、互いに異なる指数値を持つ MATRIX_A と MATRIX_B に対して Strassen の方式を使用すると、不正確な結果が出る場合があります。

MINDIM の値の意味は次のとおりです。

<=0
Strassen アルゴリズムをまったく使用しません。これはデフォルトです。

1 将来の利用に備えて予約されています。

>1 引き数配列内のすべての次元の最小のエクステントがこの値以上であれば Strassen アルゴリズムを再帰的に適用します。**MINDIM** の最適値は、マシンの構成、使用可能なメモリー、そして配列の大きさ、タイプ、および kind タイプによって異なるため、最適なパフォーマンスを得るにはこの値をさまざまに変更して試してみなければなりません。

デフォルトでは、**MATMUL** は、マトリックス乗算の従来の O(N**3) 方式を採用します。

libpthreads.a ライブラリーをリンクすると、O(N**2.81) Strassen 方式の Winograd 変分を次の条件下で採用します。

1. MATRIX_A と MATRIX_B は、両方とも整数か、または複素数であり、同じ kind タイプを持ちます。
2. プログラムは、エクステント N の正方行列のための、約 (2/3)*(N**2) 個のエレメントを保持するのに必要十分な一時ストレージを割り振ることができます。
3. MINDIM 引き数は、MATRIX_A と MATRIX_B のすべてのエクステントの最小よりも小さいか等しくなります。

IBM 拡張 の終り

最低でも 1 つの引き数のランクを 2 にしなければなりません。 MATRIX_B の唯一のまたはその最初の次元は、MATRIX_A の唯一のまたは最後の次元に等しくなければなりません。

クラス

変換関数

結果値

結果は配列になります。いずれかの引き数のランクが 1 であれば、結果のランクは 1 になります。両方の引き数のランクが 2 であれば、結果のランクは 2 になります。

結果のデータ型は、 115 ページの表 3 および 122 ページの表 4 に記載されている規則に従って、引き数のデータ型によって異なります。

MATRIX_A と MATRIX_B のデータ型が数値の場合は、結果の配列エレメントは次のようになります。

エレメントの値 (i,j) = SUM((MATRIX_A の行 i) * (MATRIX_B の列 j))

MATRIX_A と MATRIX_B のデータ型が論理タイプの場合は、結果の配列エレメントは次のようになります。

エレメントの値 (i,j) = ANY((MATRIX_A の行 i) .AND. (MATRIX_B の列 j))

例

```
! A is the array  | 1 2 3 |, B is the array | 7 10 |
!                 | 4 5 6 |                 | 8 11 |
!                                           | 9 12 |
   RES = MATMUL(A, B)
! The result is |  50   68 |
!               | 122  167 |
```

IBM 拡張

```
! HUGE_ARRAY and GIGANTIC_ARRAY in this example are
! large arrays of real or complex type, so the operation
! might be faster with the Strassen algorithm.

   RES = MATMUL(HUGE_ARRAY, GIGANTIC_ARRAY, MINDIM=196)
```

IBM 拡張 の終り

関連情報

IBM 拡張

-qessl コンパイラー・オプションが使用されている場合、コンパイラーは Fortran 実行時ライブラリーではなく、 ESSL ライブラリーを使用しようとします。詳細については、「ユーザーズ・ガイド」を参照してください。

マトリックス乗算に対する Strassen の方式の数値的安定度については、以下の資料に記述されています。

「Exploiting Fast Matrix Multiplication Within the Level 3 BLAS」、 Nicholas J. Higham, *ACM Transactions on Mathematical Software*, Vol. 16, No. 4, December 1990.

「GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm」、Douglas, C. C., Heroux, M., Slishman, G., and Smith, R. M., *Journal of Computational Physics*, Vol. 110, No. 1, January 1994, pages 1-10.

IBM 拡張 の終り

MAX (A1, A2, A3, ...)

最大値を求めます。

引き数タイプおよび属性

- **A3, ...** は、オプションの引き数です。その配列自体がオプションの仮引き数である配列は、呼び出しプロシージャー内にない場合は、オプションの引き数としてこの関数に渡してはなりません。
- 引き数はすべて、同じタイプ (整数または実数) と同じ kind 型付きパラメーターを持っていなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

引き数と同じです。 (特定の関数の中には、特定のタイプの結果を戻すものもあります。)

結果値

結果の値は、最大の引き数の値になります。

例

MAX (-9.0, 7.0, 2.0) は値 7.0 を持ちます。

MAX (10, 3, A) を計算する場合は、PRESENT(A) が呼び出しプロシージャー内で真でなければなりません。この A は、呼び出しプロシージャー内のオプションの配列引き数です。

特定名	引き数タイプ	結果タイプ	引き数渡し
AMAX0	任意の整数 **1**	デフォルトの実数	なし
AMAX1	デフォルトの実数	デフォルトの実数	なし
DMAX1	倍精度実数	倍精度実数	なし
QMAX1	REAL(16)	REAL(16)	なし
MAX0	任意の整数 **1**	引き数と同じ	なし
MAX1	任意の実数 **2**	デフォルトの整数	なし

注:

1. IBM 拡張: デフォルト以外の整数の引き数を指定するための機能。
2. IBM 拡張: デフォルト以外の実引き数を指定するための機能。

MAXEXPONENT (X)

引き数と同じタイプおよび kind 型付きパラメーターの数を表すモデル内の最大指数を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

IBM 拡張

結果は次のとおりです。

```
   type            MAXEXPONENT
----------         -----------
  real(4)               128
  real(8)              1024
  real(16)             1024
```

IBM 拡張 の終り

例

IBM 拡張

```
MAXEXPONENT(X) = 128 for X of type real(4).
```

544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

MAXLOC(ARRAY, DIM, MASK) または MAXLOC(ARRAY, MASK)

マスクの値が真になるすべてのエレメントの最大値を持つ配列の最初のエレメントを、次元に沿って見つけます。 MAXLOC は、正の整数を用いているエレメントの位置を参照できる指標を戻します。

引き数タイプおよび属性

ARRAY 整数または実数タイプの配列です。

Fortran 95

DIM (オプション)
$1 \leq DIM \leq rank(ARRAY)$ の範囲にあるスカラー整数です。

Fortran 95 の終り

MASK (オプション)
形状が ARRAY と整合している論理タイプの配列です。これが存在しないと、デフォルトのマスク評価は `.TRUE.` になります。つまり、配列全体が評価されます。

クラス

変換関数

結果タイプおよび属性

DIM が存在しない場合、結果はランク 1 の整数配列で、ARRAY のランクに等しいサイズを持ちます。 DIM が存在する場合、結果はランク rank(ARRAY)-1 の整数配列で、その形状は $(s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ となります。この場合 n は ARRAY のランクを表しています。

最大値が存在しない場合 (おそらく、配列がゼロにサイズ決定されているか、または、マスク配列がすべて `.FALSE.` の値を持っているか、または、DIM 引き数がないのが原因) は、戻り値はサイズがゼロで 1 次元のエンティティーです。 DIM が存在する場合、結果の形状は ARRAY のランクによって決まります。

結果値

結果は ARRAY の、マスクされた最大エレメントの位置の添え字を示します。複数のエレメントがこの最大値に等しいと、この関数は (配列エレメント順に) 最初の位置を検出します。 DIM が指定されると、結果は次元の各ベクトルに沿った最大マスク・エレメントの位置を示します。

Fortran 95

DIM と MASK はどちらもオプションであるため、引き数のさまざまな組み合わせが可能になります。 **-qintlog** オプションが 2 つの引き数を指定すると、2 番目の引き数は次のうちの 1 つを参照します。

- MASK。タイプが整数、論理、バイト、タイプなしの配列の場合。
- DIM。タイプが整数、バイト、またはタイプなしのスカラーの場合。

• MASK。タイプが論理のスカラーの場合。

DIM 引き数を追加すると、XL Fortran バージョン 3 からの動作が修正されます。

Fortran 95 の終り

例

```
! A is the array  |  4  9  8 -8 |
!                 |  2  1 -1  5 |
!                 |  9  4 -1  9 |
!                 | -7  5  7 -3 |

! Where is the largest element of A?
       RES = MAXLOC(A)
! The result is | 3 1 | because 9 is located at A(3,1).
! Although there are other 9s, A(3,1) is the first in
! column-major order.

! Where is the largest element in each column of A
! that is less than 7?
       RES = MAXLOC(A, DIM = 1, MASK = A .LT. 7)
! The result is | 1 4 2 2 | because these are the corresponding
! row locations of the largest value in each column
! that are less than 7 (the values being 4,5,-1,5).
```

配列の定義された上限と下限に関係なく、MAXLOC は下限の指標を '1' として判別します。MAXLOC および MINLOC 指標は両方とも正の整数を使用します。実指数を検出するには次のように行います。

```
       INTEGER B(-100:100)
! Maxloc views the bounds as (1:201)
! If the largest element is located at index '-49'
       I = MAXLOC(B)
! Will return the index '52'
! To return the exact index for the largest element, insert:
       INDEX = LBOUND(B) - 1 + I
! Which is:  INDEX = (-100) - 1 + 52 = (-49)
       PRINT*, B(INDEX)
```

MAXVAL(ARRAY, DIM, MASK) または MAXVAL(ARRAY, MASK)

MASK の真のエレメントに該当する次元に沿った配列内のエレメントの最大値を戻します。

引き数タイプおよび属性

ARRAY
整数または実数タイプの配列です。

DIM (オプション)
1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。

MASK (オプション)
: 形状が ARRAY に準拠している論理タイプの配列またはスカラーです。これが存在しないと、配列全体が評価されます。

クラス

変換関数

結果値

結果は、ランクが rank(ARRAY)-1 の配列で、データ型は ARRAY と同じです。 DIM が脱落している場合、または ARRAY のランクが 1 の場合は、結果はスカラーになります。

DIM が指定されている場合は、次元 DIM の各ベクトルに沿って MASK によって指定された条件を満たす、すべてのエレメントの最小値が結果の値の個々のエレメントに含まれます。結果の中の配列エレメントの添え字は (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) で、この n は ARRAY のランクで、 DIM は DIM で指定された次元です。

DIM が指定されていないと、関数はすべての適用可能なエレメントの最大値を戻します。

ARRAY がゼロにサイズ指定され、マスク配列がすべて .FALSE. の値をとったとき、結果の値は ARRAY と同じタイプおよび kind タイプをとる最大絶対値の負の数です。

Fortran 95

DIM と MASK はどちらもオプションであるため、引き数のさまざまな組み合わせが可能になります。 **-qintlog** オプションが 2 つの引き数を指定すると、2 番目の引き数は次のうちの 1 つを参照します。

- MASK。タイプが整数、論理、バイト、タイプなしの配列の場合。
- DIM。タイプが整数、バイト、またはタイプなしのスカラーの場合。
- MASK。タイプが論理のスカラーの場合。

Fortran 95 の終り

例

```
! A is the array  | -41  33 25 |
!                 |  12 -61 11 |

! What is the largest value in the entire array?
      RES = MAXVAL(A)
! The result is 33

! What is the largest value in each column?
```

```
       RES = MAXVAL(A, DIM=1)
! The result is | 12 33 25 |

! What is the largest value in each row?
       RES = MAXVAL(A, DIM=2)
! The result is | 33 12 |

! What is the largest value in each row, considering only
! elements that are less than 30?
       RES = MAXVAL(A, DIM=2, MASK = A .LT. 30)
! The result is | 25 12 |
```

MERGE (TSOURCE, FSOURCE, MASK)

2 つの値の間、または 2 つの配列内の対応するエレメント間で選択します。論理マスクは、個々の結果エレメントを最初の引き数からとるか、2 番目の引き数からとるかを決定します。

引き数タイプおよび属性

TSOURCE
マスク内の対応するエレメントが真である場合に使用するソース配列です。これは、任意のデータ型の式です。

FSOURCE
マスク内の対応するエレメントが偽である場合に使用するソース配列です。これは、tsource と同じデータ型と型付きパラメーターを持っていなければなりません。この形状は、tsource に準拠しています。

MASK
形状が TSOURCE および FSOURCE に準拠している論理式です。

クラス

エレメント型関数

結果値

結果は、TSOURCE および FSOURCE と同じ形状とデータ型を持ちます。

結果内の個々のエレメントの場合、TSOURCE からとられる (真の場合) か、それとも FSOURCE からとられる (偽の場合) かは、MASK 内の対応するエレメントの値によって決まります。

例

```
! TSOURCE is | A D G |, FSOURCE is | a d g |,
!            | B E H |             | b e h |
!            | C F I |             | c f i |
!
! and MASK is the array | T T T |
!                       | F F F |
!                       | F F F |
```

```
! Take the top row of TSOURCE, and the remaining elements
! from FSOURCE.
      RES = MERGE(TSOURCE, FSOURCE, MASK)
! The result is  | A D G |
!                | b e h |
!                | c f i |

! Evaluate IF (X .GT. Y) THEN
!              RES=6
!           ELSE
!              RES=12
!           END IF
! in a more concise form.
      RES = MERGE(6, 12, X .GT. Y)
```

MIN (A1, A2, A3, ...)

最小値を求めます。

引き数タイプおよび属性

- **A3, ...** は、オプションの引き数です。その配列自体がオプションの仮引き数である配列は、呼び出しプロシージャー内にない場合は、オプションの引き数としてこの関数に渡してはなりません。
- 引き数はすべて、同じタイプ (整数または実数) と同じ kind 型付きパラメーターを持っていなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

引き数と同じです。 (特定の関数の中には、特定のタイプの結果を戻すものもあります。)

結果値

結果の値は、最小の引き数の値になります。

例

MIN (-9.0, 7.0, 2.0) は値 -9.0 を持ちます。

MIN (10, 3, A) を計算する場合は、PRESENT(A) が呼び出しプロシージャー内で真でなければなりません。この A は、呼び出しプロシージャー内のオプションの配列引き数です。

特定名	引き数タイプ	結果タイプ	引き数渡し
AMIN0	任意の整数	デフォルトの実数	なし
AMIN1	デフォルトの実数	デフォルトの実数	なし
DMIN1	倍精度実数	倍精度実数	なし
QMIN1	REAL(16)	REAL(16)	なし
MIN0	任意の整数	引き数と同じ	なし
MIN1	任意の実数	デフォルトの整数	なし

MINEXPONENT (X)

引き数と同じタイプおよび kind 型付きパラメーターの数を表すモデル内の最小 (ほとんどが負) 指数を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

IBM 拡張

結果は次のとおりです。

```
   type           MINEXPONENT
----------        -----------
  real(4)            - 125
  real(8)            -1021
  real(16)            -968
```

IBM 拡張 の終り

例

IBM 拡張

```
MINEXPONENT(X) = -125 for X of type real(4).
```

544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

MINLOC(ARRAY, DIM, MASK) または MINLOC(ARRAY, MASK)

マスクの値が真になるすべてのエレメントの最小値を持つ配列の最初のエレメントを、次元に沿って見つけます。 MINLOC は、正の整数を用いているエレメントの位置を参照できる指標を戻します。

引き数タイプおよび属性

ARRAY 整数または実数タイプの配列です。

Fortran 95

DIM (オプション)
1≤DIM≤*n* の範囲にあるスカラー整数で、*n* は、ARRAY のランクを表しています。

Fortran 95 の終り

MASK (オプション)
形状が ARRAY と整合している論理タイプの配列です。これが存在しないと、デフォルトのマスク評価は `.TRUE.` になります。つまり、配列全体が評価されます。

クラス

変換関数

結果タイプおよび属性

DIM が存在しない場合、結果はランク 1 の整数配列で、ARRAY のランクに等しいサイズを持ちます。 DIM が存在する場合、結果はランク rank(ARRAY)-1 の整数配列で、その形状は $(s_1, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n)$ となります。この場合 n は ARRAY のランクを表しています。

最小値が存在しない場合 (おそらく、配列がゼロにサイズ決定されているか、または、マスク配列がすべて `.FALSE.` の値を持っているか、または、DIM 引き数がないのが原

因) は、戻り値はサイズがゼロで 1 次元のエンティティーです。 DIM が存在する場合、結果の形状は ARRAY のランクによって決まります。

結果値

結果は ARRAY の、マスクされた最小エレメントの位置の添え字を示します。複数のエレメントがこの最小値に等しいと、この関数は (配列エレメント順に) 最初の位置を検出します。 DIM が指定されると、結果は次元の各ベクトルに沿った最小マスク・エレメントの位置を示します。

Fortran 95

DIM と MASK はどちらもオプションであるため、引き数のさまざまな組み合わせが可能になります。 **-qintlog** オプションが 2 つの引き数を指定すると、2 番目の引き数は次のうちの 1 つを参照します。

- MASK。タイプが整数、論理、バイト、タイプなしの配列の場合。
- DIM。タイプが整数、バイト、またはタイプなしのスカラーの場合。
- MASK。スカラーまたは論理タイプである場合。

DIM 引き数を追加すると、XL Fortran バージョン 3 からの動作が修正されます。

Fortran 95 の終り

例

```
! A is the array  |  4  9  8 -8 |
!                 |  2  1 -1  5 |
!                 |  9  4 -1  9 |
!                 | -7  5  7 -3 |

! Where is the smallest element of A?
       RES = MINLOC(A)
! The result is | 1 4 | because -8 is located at A(1,4).

! Where is the smallest element in each row of A that
! is not equal to -7?
       RES = MINLOC(A, DIM = 2, MASK = A .NE. -7)
! The result is | 4 3 3 4 | because these are the
! corresponding column locations of the smallest value
! in each row not equal ! to -7 (the values being
! -8,-1,-1,-3).
```

配列の定義された上限と下限に関係なく、MINLOC は下限の指標を ’1’ として判別します。 MAXLOC および MINLOC 指標は両方とも正の整数を使用します。実際の指標を見つける方法を次に示します。

```
       INTEGER B(-100:100)
! Minloc views the bounds as (1:201)
! If the smallest element is located at index ’-49’
```

```
       I = MINLOC(B)
! Will return the index '52'
! To return the exact index for the smallest element, insert:
       INDEX = LBOUND(B) - 1 + I
! Which is:  INDEX = (-100) - 1 + 52 = (-49)
       PRINT*, B(INDEX)
```

MINVAL(ARRAY, DIM, MASK) または MINVAL(ARRAY, MASK)

MASK の真のエレメントに該当する次元に沿った配列内のエレメントの最小値を戻します。

引き数タイプおよび属性

ARRAY
整数または実数タイプの配列です。

DIM (オプション)
1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。

MASK (オプション)
形状が ARRAY に準拠している論理タイプの配列またはスカラーです。これが存在しないと、配列全体が評価されます。

クラス

変換関数

結果値

結果は、ランクが rank(ARRAY)-1 の配列で、データ型は ARRAY と同じです。 DIM が脱落している場合、または ARRAY のランクが 1 の場合は、結果はスカラーになります。

DIM が指定されている場合は、次元 DIM の各ベクトルに沿って MASK によって指定された条件を満たす、すべてのエレメントの最小値が結果の値の個々のエレメントに含まれます。結果の中の配列エレメントの添え字は (s_1, s_2, ..., $s_{(DIM-1)}$, $s_{(DIM+1)}$, ..., s_n) で、この n は ARRAY のランクで、 DIM は DIM で指定された次元です。

DIM が指定されていないと、関数はすべての適用可能なエレメントの最小値を戻します。

ARRAY がゼロにサイズ指定され、マスク配列がすべて .FALSE. の値をとったとき、結果の値は ARRAY と同じタイプおよび kind タイプをとる最大絶対値の正の数です。

Fortran 95

DIM と MASK はどちらもオプションであるため、引き数のさまざまな組み合わせが可能になります。 **-qintlog** オプションが 2 つの引き数を指定すると、2 番目の引き数は次のうちの 1 つを参照します。

- MASK。タイプが整数、論理、バイト、タイプなしの配列の場合。
- DIM。タイプが整数、バイト、またはタイプなしのスカラーの場合。
- MASK。タイプが論理のスカラーの場合。

Fortran 95 の終り

例

```
! A is the array  | -41  33 25 |
!                 |  12 -61 11 |

! What is the smallest element in A?
      RES = MINVAL(A)
! The result is -61

! What is the smallest element in each column of A?
      RES = MINVAL(A, DIM=1)
! The result is | -41 -61 11 |

! What is the smallest element in each row of A?
      RES = MINVAL(A, DIM=2)
! The result is | -41 -61 |

! What is the smallest element in each row of A,
! considering only those elements that are
! greater than zero?
      RES = MINVAL(A, DIM=2, MASK = A .GT.0)
! The result is | 25 11 |
```

MOD (A, P)

剰余関数です。

引き数タイプおよび属性

A タイプは整数または実数でなければなりません。

P

タイプおよび kind 型付きパラメーターは A と同じでなければなりません。

IBM 拡張

コンパイラー・オプション -qport=mod が指定されている場合、 kind 型付き

パラメーターを別のものにできます。

IBM 拡張 の終り

クラス

エレメント型関数

結果タイプおよび属性

A と同じです。

結果値

- P ≠ 0 の場合は、結果の値は A - INT(A/P) * P になります。
- P = 0 の場合は、結果は不定になります。

例

MOD (3.0, 2.0) は値 1.0 を持ちます。
MOD (8, 5) は値 3 を持ちます。
MOD (-8, 5) は値 -3 を持ちます。
MOD (8, -5) は値 3 を持ちます。
MOD (-8, -5) は値 -3 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
MOD	任意の整数	引き数と同じ	あり
AMOD	デフォルトの実数	デフォルトの実数	あり
DMOD	倍精度実数	倍精度実数	あり
QMOD	REAL(16)	REAL(16)	あり

注:

1. IBM 拡張: 名前を引き数として渡す機能。

関連情報

プログラムを XL Fortran に移植したときの **MOD** の別の動作については、「ユーザーズ・ガイド」の **-qport** コンパイラー・オプションを参照してください。

MODULO (A, P)

モジュロ関数です。

引き数タイプおよび属性

A タイプは整数または実数でなければなりません。

P タイプおよび kind 型付きパラメーターは A と同じでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

A と同じです。

結果値

- ケース (i): A のタイプは整数です。 P ≠ 0 の場合、MODULO (A, P) は A = Q * P + R (Q は整数) を満たす値 R を持ちます。

  P > 0 の場合は、不等式 0 ≤ R < P が成立します。

  P < 0 の場合は、P < R ≤ 0 が成立します。

  P = 0 の場合は、結果は不定になります。
- ケース (ii): A のタイプは実数です。 P ≠ 0 の場合、結果の値は A - FLOOR (A / P) * P になります。

  P = 0 の場合は、結果は不定になります。

例

MODULO (8, 5) は値 3 を持ちます。
MODULO (-8, 5) は値 2 を持ちます。
MODULO (8, -5) は値 -2 を持ちます。
MODULO (-8, -5) は値 -3 を持ちます。

MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)

あるデータ・オブジェクトから別のデータ・オブジェクトへビットのシーケンスをコピーします。

引き数タイプおよび属性

FROM タイプは整数でなければなりません。 これは、INTENT(IN) 引き数です。

FROMPOS タイプは整数で、負以外でなければなりません。これは、INTENT(IN) 引き数です。 FROMPOS + LEN は、BIT_SIZE (FROM) 以下でなければなりません。

LEN タイプは整数で、負以外でなければなりません。これは、INTENT(IN) 引き数です。

TO 整数タイプの変数で、FROM と同じ kind 型付きパラメーター値を持っていなければならず、FROM と同じ変数である場合もあります。これは、INTENT(INOUT) 引き数です。 TO は、長さ LEN のビットのシーケンスをコピーすることによって設定され、FROM の FROMPOS 位置から始まり、TO の TOPOS 位置まで続きます。 TO のその他のビットは変更されません。戻る時には、TOPOS で始まる TO の LEN ビットがエントリーで持っていた値に等しくなります。

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

TOPOS タイプは整数で、負以外でなければなりません。これは、INTENT(IN) 引き数です。 TOPOS + LEN は、BIT_SIZE (TO) 以下でなければなりません。

クラス

エレメント型サブルーチン

例

TO が初期値 6 を持っている場合は、次のステートメントの後の TO の値は 5 になります。

```
CALL MVBITS (7, 2, 2, TO, 0)
```

542 ページの『整数ビット・モデル』を参照してください。

NEAREST (X,S)

S という符号で示されている方向 (正または負の無限大方向) に、違いが最も小さい、プロセッサーが表現できる数字を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。

S タイプはゼロ以外の実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は、S と同じ符号を持つ無限大の方向にあり、X に最も近く、しかも X とは異なるマシン番号になります。

例

IBM 拡張

NEAREST (3.0, 2.0) = 3.0 + $2.0^{(-22)}$。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

NINT (A, KIND)

最も近い整数です。

引き数タイプおよび属性

A タイプは実数でなければなりません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 整数
- KIND が存在する場合は、kind 型付きパラメーターは KIND で指定されたものになります。存在しない場合は、kind 型付きパラメーターはデフォルトの整数タイプになります。

結果値

- A > 0 の場合は、NINT (A) は値 INT (A + 0.5) を持ちます。
- A ≤ 0 の場合は、NINT (A) は値 INT (A - 0.5) を持ちます。
- この値を指定された整数タイプで表現できない場合は、結果は不定になります。

例

NINT (2.789) は値 3 を持ちます。NINT (2.123) は値 2 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
NINT	デフォルトの実数	デフォルトの整数	あり
IDNINT	倍精度実数	デフォルトの整数	あり
IQNINT	REAL(16)	デフォルトの整数	あり **1**

注:

1. IBM 拡張: 名前を引き数として渡す機能。

NOT (I)

論理補数を実行します。

引き数タイプおよび属性

I タイプは整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

結果は、以下の表に従って、ビットごとに I の補数を求めることによって得られる値になります。

```
I NOT (I)
---------
1    0
0    1
```

ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

I が `01010101` というビットのストリングで表される場合は、NOT(I) の持つビットのストリングは `10101010` になります。 542 ページの『整数ビット・モデル』を参照してください。

特定名	引き数タイプ	結果タイプ	引き数渡し
NOT	任意の整数	引き数と同じ	あり **1**

注:

1. IBM 拡張

NULL(MOLD)

Fortran 95

この関数は、ポインターを戻すか、または構造体コンストラクターの未割り振りの割り振り可能コンポーネントを指定します。ポインターの関連付け状況は、解除されます。

次のいずれかで使用する場合は、**MOLD** 引き数を指定しないでこの関数を使用してください。

- 宣言のオブジェクトの初期化
- コンポーネントのデフォルトの初期化
- **DATA** ステートメントで
- **STATIC** ステートメントで

次のいずれかで使用する場合は、**MOLD** 引き数を指定しても、しなくてもこの関数を使用できます。

- **PARAMETER** 属性で
- ポインター割り当ての右辺で
- 構造体コンストラクターで
- 実引き数として

引き数タイプおよび属性

MOLD (オプション)
: ポインターでなければなりませんが、どんなタイプでもかまいません。ポインターの関連付け状況は、未定義、関連解除、または関連済みのいずれでもかまいません。 **MOLD** 引き数に、関連済みの関連付け状況がある場合、ターゲットは未定義になることがあります。

クラス

変換関数

結果タイプおよび属性

MOLD が存在する場合、ポインターのタイプ、型付きパラメーター、およびランクは **MOLD** と同じです。**MOLD** が存在しない場合、エンティティーのタイプ、型付きパラメーター、およびランクは以下のようになります。

- ポインター割り当てでは、左辺のポインターと同じ。
- 宣言でオブジェクトを初期化するときは、オブジェクトと同じ。

- コンポーネントのデフォルトの初期化では、コンポーネントと同じ。
- 構造体コンストラクターでは、対応するコンポーネントと同じ。
- 実引き数としては、対応する仮引き数と同じ。
- **DATA** ステートメントでは、対応するポインター・オブジェクトと同じ。
- **STATIC** ステートメントでは、対応するポインター・オブジェクトと同じ。

結果値

結果は、関連付け状況が関連解除であるポインターか、または未割り振りの割り振り可能エンティティーになります。

例

```
! Using NULL() as an actual argument.
INTERFACE
   SUBROUTINE FOO(I, PR)
      INTEGER I
      REAL, POINTER:: PR
   END SUBROUTINE FOO
END INTERFACE

CALL FOO(5, NULL())
```

Fortran 95 の終り

NUM_PARTHDS()

IBM 拡張

プログラムの実行中にランタイムが作成する Fortran の並列スレッドの数を戻します。この値は、**PARTHDS** 実行時オプションを使って設定します。ユーザーが **PARTHDS** 実行時オプションを設定しないと、実行時は **PARTHDS** にデフォルトの値を設定します。これを行うにあたって、ランタイムはオプションの設定時に次の要素を考慮することがあります。

- マシンにあるプロセッサーの数
- 実行時オプション **USRTHDS** に指定されている値

クラス

照会関数

結果値

デフォルトのスカラー整数

コンパイラー・オプション **-qsmp** が指定されていないと、**NUM_PARTHDS** は常に値 1 を戻します。

例

```
I = NUM_PARTHDS()
IF (I == 1) THEN
   CALL SINGLE_THREAD_ROUTINE()
ELSE
   CALL MULTI_THREAD_ROUTINE()
```

特定名	引き数タイプ	結果タイプ	引き数渡し
NUM_PARTHDS	デフォルトのスカラー整数	デフォルトのスカラー整数	なし

関連情報

「ユーザーズ・ガイド」の『**parthds** 実行時オプション』および『**XLSMPOPTS** 実行時オプション』の項を参照してください。

IBM 拡張 の終り

NUMBER_OF_PROCESSORS (DIM)

IBM 拡張

非 HPF プログラムの場合、値が常に 1 である、デフォルトの整数タイプのスカラーを戻します。この値は、プログラムが使用できる分散メモリー・ノードの数を参照し、常に 1 です。これにより、HPF および非 HPF 環境向けに作成されたプログラム間での下位互換性が保証されます。

引き数タイプおよび属性

DIM (オプション)

スカラー整数で、値 1 (プロセッサー配列のランク) を持っていなければなりません。

クラス

システム照会関数

結果タイプおよび属性

非 HPF プログラムの場合、常に 1 の値を持つデフォルトのスカラー整数になります。

例

```
I = NUMBER_OF_PROCESSORS()       ! 1
J = NUMBER_OF_PROCESSORS(DIM=1)  ! 1
```

IBM 拡張 の終り

NUM_USRTHDS()

IBM 拡張

プログラムの実行中にユーザーが明示的に作成するスレッドの数を戻します。この値は、**USRTHDS** 実行時オプションを使って設定します。

クラス

照会関数

結果値

デフォルトのスカラー整数

値を **USRTHDS** 実行時オプションを使って明示的に指定しないと、デフォルトの値は 0 になります。

特定名	引き数タイプ	結果タイプ	引き数渡し
NUM_USRTHDS	デフォルトの スカラー整数	デフォルトの スカラー整数	なし

関連情報

「ユーザーズ・ガイド」の『**usrthds** 実行時オプション』および『**XLSMPOPTS** 実行時オプション』を参照してください。

IBM 拡張 の終り

PACK (ARRAY, MASK, VECTOR)

配列からいくつかの、または全部のエレメントをとり、マスクの制御下でそれらをパックして 1 次元配列にします。

引き数タイプおよび属性

ARRAY ソース配列で、エレメントは結果の一部になります。どんなデータ型でも持つことができます。

MASK タイプは論理タイプでなければならず、ARRAY と適合可能でなければなりません。これは、ソース配列からどのエレメントがとられるかを判別します。それがスカラーである場合は、その値は ARRAY 内のすべてのエレメントに適用されます。

VECTOR (オプション)
埋め込み配列で、マスクによって選択されたエレメントの数が十分でない場合に結果を埋め込むのに、この配列のエレメントが使用されます。これは、ARRAY と同じデータ型と型付きパラメーターで、少なくとも MASK 内の真の値と同数のエレメントを持つ 1 次元配列です。 MASK が .TRUE. の値を持つスカラーである場合は、VECTOR は最低でも ARRAY 内の配列エレメントと同数のエレメントを持っていなければなりません。

クラス

変換関数

結果値

結果は、ARRAY と同じデータ型を持つ 1 次元配列になります。

結果のサイズは、オプションの引き数によって次のように異なります。

- VECTOR が指定されている場合は、結果の配列のサイズは VECTOR のサイズに等しくなります。
- それ以外の場合には、MASK が .TRUE. の値を持つスカラーであれば、MASK 内の真の配列エレメントの数か、または、ARRAY 内のエレメントの数に等しくなります。

ARRAY 内の配列エレメントが配列エレメントの順に取り出され、結果を形成します。MASK 内の対応する配列エレメントが .TRUE. である場合は、ARRAY からのエレメントが結果の終わりに置かれます。

エレメントが結果内に空のままで残る (VECTOR が存在し、 mask 内の .TRUE. 値よりも多くのエレメントを持っているのが原因) と、結果内の残りのエレメントは VECTOR からの対応する値に設定されます。

例

```
! A is the array  |  0  7  0 |
!                 |  1  0  3 |
!                 |  4  0  0 |

! Take only the non-zero elements of this sparse array.
! If there are less than six, fill in -1 for the rest.
RES = PACK(A, MASK= A .NE. 0, VECTOR=(/-1,-1,-1,-1,-1,-1/)
! The result is (/ 1, 4, 7, 3, -1, -1 /).
```

```
! Elements 1, 4, 7, and 3 are taken in order from A
! because the value of MASK is true only for these
! elements. The -1s are added to the result from VECTOR
! because the length (6) of VECTOR exceeds the number
! of .TRUE. values (4) in MASK.
```

PRECISION (X)

引き数と同じ kind 型付きパラメーターを持つ実数を表すモデル内の小数部精度を戻します。

引き数タイプおよび属性

X タイプは実数または複素数でなければなりません。スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

結果は次のようになります。

```
INT( (DIGITS(X) - 1) * LOG10(2) )
```

IBM 拡張

したがって、

```
      Type               Precision
--------------------     ---------
 real(4) , complex(4)        6
 real(8) , complex(8)       15
real(16) , complex(16)      31
```

IBM 拡張 の終り

例

IBM 拡張

PRECISION (X) = INT((24 - 1) * LOG10(2.)) = INT(6.92 ...) = 6 は、実数タイプ (4)

の X です。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

PRESENT (A)

オプションの引き数が存在するかどうかを判別します。存在しない場合は、それをオプションの引き数として別のプロシージャーに渡すか、または、それを引き数として PRESENT に渡すことしかできません。

引き数タイプおよび属性

A **PRESENT** 関数参照があるプロシージャー内でアクセス可能なオプションの仮引き数の名前です。

クラス

照会関数

結果タイプおよび属性

デフォルトの論理スカラー

結果値

実引き数が存在する場合 (つまり、指定された仮引き数内の現行プロシージャーに渡された場合) は、結果は .TRUE. になり、それ以外の場合は .FALSE. になります。

例

```
      SUBROUTINE SUB (X, Y)
        REAL, OPTIONAL :: Y
        IF (PRESENT (Y)) THEN
! In this section, we can use y like any other variable.
           X = X + Y
           PRINT *, SQRT(Y)
        ELSE
! In this section, we cannot define or reference y.
           X = X + 5
! We can pass it to another procedure, but only if
! sub2 declares the corresponding argument as optional.
           CALL SUB2 (Z, Y)
        ENDIF
      END SUBROUTINE SUB
```

関連情報

427 ページの『OPTIONAL』

PROCESSORS_SHAPE ()

IBM 拡張

ゼロにサイズ指定された配列を戻します。

クラス

システム照会関数

結果タイプおよび属性

サイズがプロセッサー配列のランクと等しい、ランク 1 のデフォルトの整数配列。単一プロセッサー環境では、結果はゼロ・サイズのベクトルになります。

結果値

結果の値は、プロセッサー配列の形状になります。

例

```
I=PROCESSORS_SHAPE()
! Zero-sized vector of type default integer
```

IBM 拡張 の終り

PRODUCT(ARRAY, DIM, MASK) または PRODUCT(ARRAY, MASK)

配列全体内のすべてのエレメントを乗算するか、次元に沿ったすべてのベクトルから選択したエレメントを乗算します。

引き数タイプおよび属性

ARRAY
数値データ型を持つ配列です。

DIM (オプション)
1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。

MASK (オプション)
形状が ARRAY に準拠している論理式です。 MASK がスカラーである場合は、その値は ARRAY 内のすべてのエレメントに適用されます。

クラス

変換関数

結果値

DIM が存在する場合は、結果はランク rank(ARRAY)-1 の配列で、 ARRAY と同じデータ型になります。 DIM が脱落している場合、または MASK のランクが 1 の場合は、結果はスカラーになります。

結果は、以下のいずれかの方式で計算されます。

方式 1: ARRAY だけが指定されている場合は、結果はその配列エレメント全部の積になります。 ARRAY がゼロ・サイズ配列である場合は、結果はゼロに等しくなります。

方式 2: ARRAY と MASK が両方とも指定されている場合は、結果は MASK 内の対応する真の配列エレメントを持つ ARRAY のそれらの配列エレメントの積になります。値が .TRUE. であるエレメントを MASK が持っていない場合は、結果は 1 に等しくなります。

方式 3: DIM も指定されて、ARRAY のランクが 1 の場合、結果は MASK 内の対応する .TRUE. 配列エレメントを持つ ARRAY のすべてのエレメントの積に等しいスカラーです。

DIM も指定されて、ARRAY のランクが 1 より大きい場合、結果は次元 DIM が除去されている新しい配列です。それぞれの新しい配列エレメントは、ARRAY 中の対応するベクトルからのエレメントの積です。そのベクトルのインデックス値は、DIM を除くすべての次元で、出力エレメントの指標値と一致します。出力エレメントは MASK 内の対応する.TRUE. 配列エレメントを持つこれらのベクトル・エレメントの積です。

Fortran 95

DIM と MASK はどちらもオプションであるため、引き数のさまざまな組み合わせが可能になります。 **-qintlog** オプションが 2 つの引き数を指定すると、2 番目の引き数は次のうちの 1 つを参照します。

- MASK。タイプが整数、論理、バイト、タイプなしの配列の場合。
- DIM。タイプが整数、バイト、またはタイプなしのスカラーの場合。
- MASK。タイプが論理のスカラーの場合。

Fortran 95 の終り

例

方式 1:

```
! Multiply all elements in an array.
       RES = PRODUCT( (/2, 3, 4/) )
! The result is 24 because (2 * 3 * 4) = 24.
```

```
! Do the same for a two-dimensional array.
       RES = PRODUCT( (/2, 3, 4/), (/4, 5, 6/) )
! The result is 2880.  All elements are multiplied.
```

方式 2:

```
! A is the array (/ -3, -7, -5, 2, 3 /)
! Multiply all elements of the array that are > -5.
       RES = PRODUCT(A, MASK = A .GT. -5)
! The result is -18 because (-3 * 2 * 3) = -18.
```

方式 3:

```
! A is the array | -2  5  7 |
!                |  3 -4  3 |
! Find the product of each column in A.
       RES = PRODUCT(A, DIM = 1)
! The result is | -6 -20 21 | because (-2 * 3) =  -6
!                                     ( 5 * -4 ) = -20
!                                     ( 7 *  3 ) = 21

! Find the product of each row in A.
       RES = PRODUCT(A, DIM = 2)
! The result is | -70 -36 |
! because (-2 *  5 * 7) = -70
!          (3 * -4 * 3) = -36

! Find the product of each row in A, considering
! only those elements greater than zero.
       RES = PRODUCT(A, DIM = 2, MASK = A .GT. 0)
! The result is | 35 9 | because ( 5 * 7) = 35
!                                  (3 * 3) =  9
```

QCMPLX (X, Y)

IBM 拡張

拡張複素数タイプに変換します。

引き数タイプおよび属性

X
タイプは整数、実数、複素数のいずれかでなければなりません。

Y (オプション)
タイプは整数または実数でなければなりません。 X のタイプが複素数の場合には、Y は指定できません。

クラス

エレメント型関数

結果タイプおよび属性

タイプは拡張複素数です。

結果値

- Y が存在せず X が複素数でない場合は、Y がゼロという値を持って存在しているかのようになります。
- Y が存在せず X が複素数である場合は、Y が値 AIMAG(X) を持ち、X は値 REAL(X) を持って存在しているかのようになります。
- QCMPLX(X, Y) は、実数部分は REAL(X, KIND=16) で、虚数部分が REAL(Y, KIND=16) である複素数値を持ちます。

例

QCMPLX (-3) は値 (-3.0Q0, 0.0Q0) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
QCMPLX	REAL(16)	COMPLEX(16)	なし

関連情報

566 ページの『CMPLX (X, Y, KIND)』、 579 ページの『DCMPLX (X, Y)』を参照してください。

IBM 拡張 の終り

QEXT (A)

IBM 拡張

拡張精度実数タイプに変換します。

引き数タイプおよび属性

A タイプは整数または実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

拡張精度実数

結果値

- A のタイプが拡張精度実数の場合は、QEXT(A) = A になります。
- A のタイプが整数または実数の場合は、結果は A を拡張精度で正確に表現した値です。

例

QEXT (-3) は値 -3.0Q0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
QFLOAT	任意の整数	REAL(16)	なし
QEXT	デフォルトの実数	REAL(16)	なし
QEXTD	倍精度実数	REAL(16)	なし

IBM 拡張 の終り

RADIX (X)

引き数と同じタイプおよび kind 型付きパラメーターの数を表すモデルの基数を戻します。

引き数タイプおよび属性

X タイプは整数または実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

結果は、X と同じ kind およびタイプの数値を表すモデルの基数になります。

IBM 結果は、常に 2 です。 IBM 542 ページの『データ表示モデル』のモデルを参照してください。

RAND ()

IBM 拡張

極力使用しないでください。 0.0 以上 1.0 未満の正の実数の一様乱数を生成します。この代わりに、RANDOM_NUMBER 組み込みサブルーチンを使用してください。

クラス

なし (定義されているカテゴリーのどれにも対応しません)

結果タイプおよび属性

real(4) スカラー

関連情報

677 ページの『SRAND (SEED)』は、乱数列に対してシード値を指定するのに使用できます。

関数の結果が配列に割り当てられる場合は、すべての配列エレメントが同じ値を受け取ります。

例

以下は、RAND 関数を使用するプログラムの例です。

```
DO I = 1, 5
   R = RAND()
   PRINT *, R
ENDDO
END
```

上記のプログラムで生成される出力例は次のとおりです。

```
0.2251586914
0.8285522461
0.6456298828
0.2496948242
0.2215576172
```

この関数だけが特定名を持ちます。

IBM 拡張 の終り

RANDOM_NUMBER (HARVEST)

1 つの疑似乱数、または、0 ≤ x < 1 の範囲の一様分布からの疑似乱数の配列を戻します。

libpthreads.a ライブラリーをリンクすると、並列インプリメンテーションによる乱数発生を採用します。これにより、SMP マシン上でのパフォーマンスが向上します。使用されるスレッドの数は、**intrinheads=***num* 実行時オプションによって制御できます。

引き数タイプおよび属性

HARVEST
タイプは実数でなければなりません。 これは、INTENT(OUT) 引き数です。スカラー変数または配列変数を使用できます。 0 ≤ x < 1 の間で一様分布からの疑似乱数を含むように設定されます。

クラス

サブルーチン

例

```
REAL X, Y (10, 10)
! Initialize X with a pseudo-random number
CALL RANDOM_NUMBER (HARVEST = X)
CALL RANDOM_NUMBER (Y)
! X and Y contain uniformly distributed random numbers
```

RANDOM_SEED (SIZE, PUT, GET, GENERATOR)

RANDOM_NUMBER によって使用された疑似乱数発生ルーチンを再始動または照会します。

引き数タイプおよび属性

引き数がちょうど 1 つだけか、またはまったく存在しないかのどちらかでなければなりません。

SIZE (オプション)
スカラーで、タイプはデフォルトの整数でなければなりません。これは、INTENT(OUT) 引き数です。 8 バイト変数であるシード値を保持するのに必要なデフォルト・タイプの整数 (N) の数に設定されます。

PUT (オプション)
ランク 1 でサイズが N 以上のデフォルトの整数配列でなければなりません。これは、INTENT(IN) 引き数です。 現行の乱数発生ルーチン用のシードがここから転送されます。

GET (オプション)
ランク 1 でサイズが N 以上のデフォルトの整数配列でなければなりません。これは、INTENT(OUT) 引き数です。現行の乱数発生ルーチン用のシードがここに転送されます。

IBM 拡張

GENERATOR (オプション)
スカラーで、タイプはデフォルトの整数でなければなりません。これは、INTENT(IN) 引き数です。 この値は、以後使用する乱数発生ルーチンを決定します。値は、1 または 2 でなければなりません。

IBM 拡張 の終り

IBM 拡張

Random_seed を使用すると、2 つの乱数発生ルーチンを切り換えることができます。乱数発生ルーチン 1 がデフォルトです。各乱数発生ルーチンは、それぞれ専用のシードを維持し、通常は、最後に生成した数の次の数からサイクルを再開します。

乱数発生ルーチン 1 は、次の式により、乗算合同式法を使用します。

```
S(I+1) = ( 16807.0 * S(I) ) mod (2.0**31-1)
```

かつ

```
X(I+1) = S(I+1) / (2.0**31-1)
```

乱数発生ルーチン 1 は、 2**31-2 の乱数後に循環します。

乱数発生ルーチン 2 も、次の式により、乗算合同式法を使用します。

```
S(I+1) = ( 44,485,709,377,909.0 * S(I) )
              mod (2.0**48)
```

かつ

```
X(I+1) = S(I+1) / (2.0**48)
```

乱数発生ルーチン 2 は、(2**46) 個の乱数発生後に循環します。乱数発生ルーチン 1 がデフォルトになっていますが (以前のバージョンとの互換性を保つため)、乱数発生ルーチン 1 よりも乱数発生ルーチン 2 の方が実行が速く、循環するまでの間隔も長くなっているため、新しいプログラムでは乱数発生ルーチン 2 を使用することをお勧めします。

引き数が存在しない場合は、現行の乱数発生ルーチンのシードはデフォルト値 1d0 に設定されます。

IBM 拡張 の終り

クラス

サブルーチン

例

```
CALL RANDOM_SEED
   ! Current generator sets its seed to 1d0
CALL RANDOM_SEED (SIZE = K)
   ! Sets K = 64 / BIT_SIZE( 0 )
CALL RANDOM_SEED (PUT = SEED (1 : K))
   ! Transfer seed to current generator
CALL RANDOM_SEED (GET = OLD (1 : K))
   ! Transfer seed from current generator
```

RANGE (X)

引き数と同じ kind 型付きパラメーターを持つ整数または実数を表すモデル内の小数部指数範囲を戻します。

引き数タイプおよび属性

X タイプは整数、実数、複素数のいずれかでなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

1. 整数引き数の場合は、結果は次のようになります。

   ```
   INT( LOG10( HUGE(X) ) )
   ```

2. 実数または複素数引き数の場合は、結果は次のようになります。

   ```
   INT( MIN( LOG10( HUGE(X) ), -LOG10( TINY(X) ) ) )
   ```

IBM 拡張

したがって次のようになります。

```
Type                      RANGE
------------------------------
integer(1)                   2
integer(2)                   4
integer(4)                   9
integer(8)                  18
real(4) , complex(4)        37
real(8) , complex(8)       307
real(16) , complex(16)     291
```

IBM 拡張 の終り

例

IBM 拡張

X のタイプは real(4) です。
HUGE(X) = 0.34E+39

TINY(X) = 0.11E-37
RANGE(X) = 37

IBM 拡張 の終り

542 ページの『データ表示モデル』を参照してください。

REAL (A, KIND)

実数タイプに変換します。

引き数タイプおよび属性

A タイプは整数、実数、複素数のいずれかでなければなりません。

KIND (オプション)
スカラー整数の初期化式でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

- 実数
- ケース (i): A のタイプが整数または実数で、 KIND が存在する場合は、 kind 型付きパラメーターは KIND で指定されたものになります。 A のタイプが整数または実数で、KIND が存在しない場合は、kind 型付きパラメーターはデフォルトの実数タイプの kind 型付きパラメーターになります。
- ケース (ii): A のタイプが複素数で KIND が存在する場合は、 kind 型付きパラメーターは KIND で指定されたものになります。 A のタイプが複素数で KIND が存在しない場合は、kind 型付きパラメーターは A の kind 型付きパラメーターになります。

結果値

- ケース (i): A のタイプが整数または実数の場合は、結果は kind に依存し、A への近似値と等しくなります。
- ケース (ii): A のタイプが複素数の場合は、結果は kind に依存し、A の実数部分への近似値と等しくなります。

例

REAL (-3) は値 -3.0 を持ちます。 REAL ((3.2, 2.1)) は値 3.2 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
REAL	デフォルトの整数	デフォルトの実数	なし
FLOAT	任意の整数 1	デフォルトの実数	なし
SNGL	倍精度実数	デフォルトの実数	なし
SNGLQ	REAL(16)	デフォルトの実数	なし 2
DREAL	倍精度複素数	倍精度実数	なし 2
QREAL	COMPLEX(16)	REAL(16)	なし 2

注:

1. IBM 拡張: デフォルト以外の整数の引き数を指定するための機能。
2. IBM 拡張: 名前を引き数として渡すことができません。

REPEAT (STRING, NCOPIES)

ストリングのコピーをいくつか連結します。

引き数タイプおよび属性

STRING スカラーで、タイプは文字でなければなりません。

NCOPIES スカラーで、タイプは整数でなければなりません。 値は負であってはなりません。

クラス

変換関数

結果タイプおよび属性

NCOPIES * LENGTH(STRING) に等しい長さを持ち、STRING と同じ kind 型付きパラメーターを持つ文字スカラー。

結果値

結果の値は、STRING のコピーを NCOPIES 個だけ連結したものになります。

例

REPEAT ('H', 2) は値 'HH' を持ちます。 REPEAT ('XYZ', 0) は長さがゼロのストリングの値を持ちます。

RESHAPE (SOURCE, SHAPE, PAD, ORDER)

所定の配列のエレメントから、指定の形状の配列を構成します。

引き数タイプおよび属性

SOURCE
任意のタイプの配列で、これは結果配列に対してエレメントを与えます。

SHAPE
結果の配列の形状を定義します。これは最大 20 個のエレメントからなる整数配列で、ランクは 1、サイズは一定です。すべてのエレメントが正の整数かゼロを持ちます。

PAD (オプション)
SOURCE がもっと大きな配列に形状変更された場合に、余分な値を埋め込むのに使用されます。これは、SOURCE と同じデータ型の配列です。これが存在しない場合、あるいは、ゼロ・サイズ配列である場合、ユーザーができることは SOURCE を同じサイズまたはそれよりも小さい別の配列にすることだけです。

ORDER (オプション)
サイズが一定で、ランクが 1 の整数配列です。そのエレメントは、(1, 2, ..., SIZE(SHAPE)) の順列でなければなりません。これを使用して、通常の (1, 2, ..., rank(RESULT)) とは異なる次元順序で結果にエレメントを挿入することができます。

クラス

変換関数

結果値

結果は、形状 SHAPE を持つ配列になります。 これは、SOURCE と同じデータ型を持ちます。

SOURCE の配列エレメントは、ORDER で指定された次元の順序に従って、あるいは、ORDER が指定されていない場合は配列エレメント用の通常の順序に従って、結果内に配置されます。

SOURCE の配列エレメントの後に、PAD の配列エレメントが配列エレメント順に続き、その後にさらに PAD のコピーが、結果のエレメントがすべて設定されるまで続きます。

例

```
! Turn a rank-1 array into a 3x4 array of the
! same size.
RES= RESHAPE( (/A,B,C,D,E,F,G,H,I,J,K,L/), (/3,4/)
! The result is | A D G J |
!               | B E H K |
!               | C F I L |

! Turn a rank-1 array into a larger 3x5 array.
```

```
! Keep repeating -1 and -2 values for any
! elements not filled by the source array.
! Fill the rows first, then the columns.
RES= RESHAPE( (/1,2,3,4,5,6/), (/3,5/), &
  (/-1,-2/), (/2,1/) )
! The result is | 1  2  3  4  5  |
!               |  6 -1 -2 -1 -2 |
!               | -1 -2 -1 -2 -1 |
```

関連情報

666 ページの『SHAPE(SOURCE)』を参照してください。

RRSPACING (X)

引き数値に近いモデル番号の相対スペーシングの逆数を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は次のようになります。

ABS(FRACTION(X)) * FLOAT(RADIX(X))$^{\mathrm{DIGITS(X)}}$

例

IBM 拡張

RRSPACING (-3.0) = 0.75 * 2^{24}。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

RSHIFT (I, SHIFT)

IBM 拡張

右への論理シフトを実行します。

引き数タイプおよび属性

I　タイプは整数でなければなりません。

SHIFT　タイプは整数でなければなりません。 負以外で、BIT_SIZE(I) 以下でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

I と同じです。

結果値

- 結果は、I のビットを SHIFT 位置だけ右にシフトすることによって得られる値になります。
- 空のビットは符号ビットで埋め込まれます。
- ビットには、右から左へ 0 から BIT_SIZE(I)-1 までの番号が付けられます。

例

RSHIFT (3, 1) は結果 1 を持ちます。

RSHIFT (3, 2) は結果 0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
RSHIFT	任意の整数	引き数と同じ	あり

IBM 拡張 の終り

SCALE (X,I)

位取りされた値 `X * 2.0`^{I} を戻します。

引き数タイプおよび属性

X　タイプは実数でなければなりません。

I　タイプは整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

IBM 拡張

結果は次のように決定されます。

```
X * 2.0^I
```

SCALE (X, I) = X * (2.0^{I})

IBM 拡張 の終り

例

IBM 拡張

SCALE (4.0, 3) = 4.0 * (2^{3}) = 32.0。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

SCAN (STRING, SET, BACK)

ストリングを走査して、一連の文字内の任意の 1 文字を探します。

引き数タイプおよび属性

STRING
タイプは文字でなければなりません。

SET
タイプは STRING と同じ kind 型付きパラメーターを持つ文字でなければなりません。

BACK (オプション)
タイプは論理タイプでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

- ケース (i): BACK が存在しないか、または、値 .FALSE. を持って存在していて、 SET 内にある文字が最低 1 つ STRING に含まれていると、結果の値は、 SET 内にある STRING の左端の文字位置になります。
- ケース (ii): BACK が存在していて値 .TRUE. を持ち、 SET 内にある文字が最低 1 つ STRING に含まれている場合は、結果の値は、 SET 内にある STRING の右端の文字位置になります。
- ケース (iii): STRING の文字が SET 内にない場合、あるいは、STRING または SET の長さがゼロである場合は、結果の値はゼロになります。

例

- ケース (i): SCAN (’FORTRAN’, ’TR’) は値 3 を持ちます。
- ケース (ii): SCAN (’FORTRAN’, ’TR’, BACK = .TRUE.) は値 5 を持ちます。
- ケース (iii): SCAN (’FORTRAN’, ’BCD’) は値 0 を持ちます。

SELECTED_INT_KIND (R)

すべての整数値 n を $-10^{R} < n < 10^{R}$ で表す整数データ型の kind 型付きパラメーターの値を戻します。

引き数タイプおよび属性

R 整数タイプのスカラーでなければなりません。

クラス

変換関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

- 結果は、値 n の範囲内のすべての値 n を $-10^{R} < n < 10^{R}$ で表す整数データ型の kind 型付きパラメーターの値に等しい値になります。あるいは、そういった kind 型付きパラメーターが使用できない場合は、結果は -1 になります。
- 複数の kind 型付きパラメーターが基準を満たしている場合は、戻される値は、最小の小数部指数範囲を持つ値になります。

例

IBM 拡張

SELECTED_INT_KIND (9) は値 4 を持ちます。これは、kind タイプ 4 を持つ INTEGER が 10^{-9} から 10^{9} のすべての値を表すことができることを意味します。

IBM 拡張 の終り

関連情報

XL Fortran がサポートしている kind 型付きパラメーターについては、 27 ページの『型付きパラメーターおよび指定子』を参照してください。

SELECTED_REAL_KIND (P, R)

最低 P 桁の 10 進精度と最低 R の 10 進指数範囲を持つ実数データ型の kind 型付きパラメーターの値を戻します。

引き数タイプおよび属性

P (オプション) スカラーで、タイプは整数でなければなりません。

R (オプション) スカラーで、タイプは整数でなければなりません。

クラス

変換関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

- 結果の値は、最低 P 桁の 10 進精度 (関数 PRECISION が戻すものと同じ) と、最低 R の 10 進指数範囲 (関数 RANGE が戻すものと同じ) とを持つ実数データ型の kind 型付きパラメーターの値に等しくなります。そのような kind 型付きパラメーターがない場合には、結果は次のようになります。
  - 精度が使用できない場合は、結果は -1 になります。
  - 指数範囲が使用できない場合は、結果は -2 になります。
  - どちらも使用できない場合は、結果は -3 になります。
- 複数の kind 型付きパラメーター値が基準を満たしている場合、戻される値は、最小の小数部指数範囲を持つ値になります。ただし、そのような値がいくつかある場合には、それらの kind 値のうち最小のものが戻されます。

例

IBM 拡張

SELECTED_REAL_KIND (6, 70) は値 8 を持ちます。

IBM 拡張 の終り

関連情報

XL Fortran がサポートしている kind 型付きパラメーターについては、 27 ページの『型付きパラメーターおよび指定子』を参照してください。

SET_EXPONENT (X,I)

小数部が X のモデル表現の小数部で、指数部分が I の数字を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。

I タイプは整数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

IBM 拡張

- X = 0 の場合は、結果はゼロです。
- そうでない場合、結果は以下のようになります。

  FRACTION(X) * 2.0^{I}

IBM 拡張 の終り

例

IBM 拡張

SET_EXPONENT (10.5, 1) = 0.65625 * 2.0^{1} = 1.3125

544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

SHAPE(SOURCE)

配列またはスカラーの形状を戻します。

引き数タイプおよび属性

SOURCE
任意のデータ型の配列またはスカラーです。これは、関連解除されたポインター、割り振られていない割り振り可能オブジェクト、または想定サイズ配列であってはなりません。

クラス

照会関数

結果値

結果は、エレメントが SOURCES の形状を定義する 1 次元のデフォルトの整数配列になります。 SOURCES 内の個々の次元のエクステントは、結果配列内の対応するエレメントで戻されます。

関連情報

658 ページの『RESHAPE (SOURCE, SHAPE, PAD, ORDER)』

例

```
! A is the array  | 7 6 3 1 |
!                 | 2 4 0 9 |
!                 | 5 7 6 8 |
!
       RES = SHAPE( A )
! The result is | 3  4 | because A is a rank-2 array
! with 3 elements in each column and 4 elements in
! each row.
```

SIGN (A, B)

A の絶対値に B の符号を掛けたものを戻します。 A がゼロではない場合、その結果の符号は B の符号と同じになるため、それを使用して B が負であるか、そうでないかを判別することができます。

B を **REAL(4)** または **REAL(8)** と宣言し、B が負のゼロ値である場合、結果の符号は、**-qxlf90=signedzero** コンパイラー・オプションを指定したかしないかによって異なります。

引き数タイプおよび属性

A タイプは整数または実数でなければなりません。

B タイプおよび kind 型付きパラメーターは A と同じでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

A と同じです。

結果値

結果は *sgn**|A| です。この説明を以下に行います。

- 次のどちらかが当てはまる場合、*sgn* = -1。
  - B < 0

IBM 拡張

- B は 負のゼロ値が付いた番号 **REAL(4)** または番号 **REAL(8)** であり、ユーザーが **-qxlf90=signedzero** オプションを指定しました。

IBM 拡張 の終り

- 上記以外の場合、*sgn* = 1。

Fortran 95

Fortran 95 では、Fortran 90 では行えなかった、正の実数ゼロと負の実数ゼロをプロセッサーによって見分けることが可能です。**-qxlf90=signedzero** オプションを使用すると、Fortran 95 の動作を指定することができます (**REAL(16)** の数値のケース以外)。これによって、IEEE 標準の 2 進浮動小数点演算と整合させることができます。**-qxlf90=signedzero** は、**xlf95**、**xlf95_r**、および **xlf95_r7** 呼び出しコマンドのデフォルトです。

Fortran 95 の終り

例

SIGN (-3.0, 2.0) は値 3.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
SIGN	デフォルトの実数	デフォルトの実数	あり

特定名	引き数タイプ	結果タイプ	引き数渡し
ISIGN	任意の整数 1	引き数と同じ	あり
DSIGN	倍精度実数	倍精度実数	あり
QSIGN	REAL(16)	REAL(16)	あり 2

注:

1. IBM 拡張: デフォルト以外の整数の引き数を指定するための機能。
2. IBM 拡張: 名前を引き数として渡す機能。

関連情報

「ユーザーズ・ガイド」の『**-qxlf90** オプション』を参照してください。

SIGNAL (I, PROC)

IBM 拡張

SIGNAL プロシージャーを使用すると、プログラムは特定のオペレーティング・システム・シグナルの受信時に呼び出されるプロシージャーを指定することができます。

引き数タイプおよび属性

I
機能するシグナルの値を指定する整数です。これは、INTENT(IN) 引き数です。使用可能なシグナルの値は、C インクルード・ファイル **signal.h** に定義されます。シグナルの値のサブセットは、 Fortran インクルード・ファイル **fexcp.h** に定義されます。

PROC
指定されたシグナル (I) をプロセスが受信すると呼び出されるユーザー定義のプロシージャーを指定します。これは、INTENT(IN) 引き数です。

クラス

サブルーチン

例

```
      INCLUDE 'fexcp.h'
      INTEGER   SIGUSR1
      EXTERNAL  USRINT
! Set exception handler to produce the traceback code.
! The SIGTRAP is defined in the include file fexcp.h.
! xl__trce is a procedure in the XL Fortran
! run-time library.  It generates the traceback code.
      CALL SIGNAL(SIGTRAP, XL__TRCE)
      ...
```

```
! Use user-defined procedure USRINT to handle the signal
! SIGUSR1.
      CALL SIGNAL(SIGUSR1, USRINT)
      ...
```

関連情報

基礎となるのインプリメンテーションに関する詳細は、「*AIX Technical Reference: Base Operating System and Extensions Volume 2*」の『**signal** subroutine』を参照してください。

「ユーザーズ・ガイド」の『**-qsigtrap** オプション』には、コンパイラー・オプションを使用して **SIGTRAP** シグナル用にハンドラーを設定する方法が記載されています。

IBM 拡張 の終り

SIN (X)

サイン (正弦) 関数です。

引き数タイプおよび属性

X タイプは実数または複素数でなければなりません。 X が実数の場合は、ラジアンの値と見なされます。 X が複素数の場合は、その実数部分と虚数部分がラジアンの値と見なされます。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

これは sin(X) とほぼ同じ値になります。

例

SIN (1.0) は値 0.84147098 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
SIN	デフォルトの実数	デフォルトの実数	あり
DSIN	倍精度実数	倍精度実数	あり
QSIN	REAL(16)	REAL(16)	可 **1**
CSIN **2a**	デフォルトの複素数	デフォルトの複素数	あり

特定名	引き数タイプ	結果タイプ	引き数渡し
CDSIN 2b	倍精度複素数	倍精度複素数	可 1
ZSIN 2b	倍精度複素数	倍精度複素数	可 1
CQSIN 2b	COMPLEX(16)	COMPLEX(16)	可 1

注:

1. IBM 拡張: 名前を引き数として渡す機能。
2. X が a + bi という形式の複素数であるとすると、以下のようになります (ただし、$i = (-1)^{\frac{1}{2}}$)。
   a. abs(b) は 88.7228 以下でなければなりません (a は任意の実数値)。
   b. abs(b) は 709.7827 以下でなければなりません (a は任意の実数値)。

SIND (X)

IBM 拡張

サイン (正弦) 関数です。引き数は「度」の単位となります。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- これは sin(X) とほぼ同じ値を持ち、この X は「度」の単位の値を持ちます。

例

SIND (90.0) は値 1.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
SIND	デフォルトの実数	デフォルトの実数	あり
DSIND	倍精度実数	倍精度実数	あり
QSIND	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

SINH (X)

双曲線サイン (双曲線正弦) 関数です。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は sinh(x) に等しい値を持ちます。

例

SINH (1.0) は値 1.1752012 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
SINH **1**	デフォルトの実数	デフォルトの実数	あり
DSINH **2**	倍精度実数	倍精度実数	あり
QSINH **2** **3**	REAL(16)	REAL(16)	あり

注:

1. abs(X) は 89.4159 以下でなければなりません。
2. abs(X) は 709.7827 以下でなければなりません。
3. IBM 拡張

SIZE (ARRAY, DIM)

指定された次元に沿った配列のエクステント、または、配列内のエレメントの合計数を戻します。

引き数タイプおよび属性

ARRAY データ型の配列です。これは、スカラー、関連解除されたポインタ

ー、または割り振りされていない割り振り可能配列であってはなりません。 DIM が存在して、ARRAY のランクよりも小さい値を持っている場合には、これは想定サイズ配列になることがあります。

DIM (オプション)
: $1 \leq \text{DIM} \leq \text{rank(ARRAY)}$ の範囲内の整数スカラーです。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

結果は、次元 DIM に沿った ARRAY のエクステントに等しくなり、 DIM が指定されない場合は、ARRAY 内の配列エレメントの合計数に等しくなります。

例

```
! A is the array  | 1 -4  7 -10 |
!                 | 2  5 -8  11 |
!                 | 3  6  9 -12 |

       RES = SIZE( A )
! The result is 12 because there are 12 elements in A.

       RES = SIZE( A, DIM = 1)
! The result is 3 because there are 3 rows in A.

       RES = SIZE( A, DIM = 2)
! The result is 4 because there are 4 columns in A.
```

SIZEOF(A)

IBM 拡張

引き数のサイズをバイト単位で戻します。

引き数タイプおよび属性

A
: 以下のいずれにも当てはまらないデータ・オブジェクト。
  - Fortran 90 ポインター
  - 自動オブジェクト
  - 割り振り可能オブジェクト

- 割り振り可能コンポーネントまたは Fortran 90 ポインター・コンポーネントを持つ派生オブジェクトまたはレコード構造。
- 配列セクション
- 配列コンストラクター
- 想定形状配列
- 想定サイズ配列全体
- ゼロ・サイズ配列
- 有効範囲単位内でアクセスが不可能なコンポーネントを含む派生オブジェクトまたはレコード構造。

SIZEOF はサブプログラムに引き数として渡すことはできません。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数スカラー

結果値

引き数のサイズ (バイト単位)。

例

以下の例では、**-qintsize**=4 と想定しています。

```
INTEGER ARRAY(10)
INTEGER*8, PARAMETER :: p = 8
STRUCTURE /STR/
  INTEGER I
  COMPLEX C
END STRUCTURE
RECORD /STR/ R
CHARACTER*10 C
TYPE DTYPE
  INTEGER ARRAY(10)
END TYPE
TYPE (DTYPE) DOBJ
PRINT *, SIZEOF(ARRAY), SIZEOF (ARRAY(3)), SIZEOF(P) ! Array, array
                                                     ! element ref,
                                                     ! named constant

PRINT *, SIZEOF (R), SIZEOF(R.C)                     ! record structure
                                                     ! entity, record
                                                     ! structure
                                                     ! component

PRINT *, SIZEOF (C(2:5)), SIZEOF(C)                  ! character
```

```
                                                         ! substring,
                                                         ! character
                                                         ! variable

    PRINT *, SIZEOF (DOBJ), SIZEOF(DOBJ%ARRAY)           ! derived type
                                                         ! object, structure
                                                         ! component
```

上記のプログラムで生成される出力例は次のとおりです。

```
40   4   8
16   8
 4  10
40  40
```

関連情報

-qintsize コンパイラー・オプションの詳細については、「ユーザーズ・ガイド」を参照してください。

IBM 拡張 の終り

SPACING (X)

引き数値に近いモデル番号の絶対スペーシングを戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

X が 0 でない場合、結果は次のようになります。

$2.0^{\mathrm{EXPONENT(X)\ -\ DIGITS(X)}}$

X が 0 の場合、結果は TINY(X) の結果値と同じです。

例

IBM 拡張

SPACING (3.0) = 2.0^{2-24} = $2.0^{(-22)}$。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

SPREAD (SOURCE, DIM, NCOPIES)

その次元に沿った既存のエレメントをコピーすることによって、追加の次元内の配列を複製します。

引き数タイプおよび属性

SOURCE
配列またはスカラーを使用できます。どんなデータ型でも持つことができます。 SOURCE のランクは、最大値 19 を持ちます。

DIM 1 ≤ DIM ≤ rank(SOURCE)+1 の範囲内の整数スカラーです。他のほとんどの配列組み込み関数とは異なり、**SPREAD** は DIM 引き数が必要です。

NCOPIES
整数スカラーです。これは、結果に追加される余分の次元のエクステントになります。

クラス

変換関数

結果タイプおよび属性

結果は、ランクが rank(SOURCE)+1 で、タイプおよび型付きパラメーターが source と同じです。

結果値

SOURCE がスカラーの場合、結果は NCOPIES 個のエレメントを持ち、そのおのおのが値 SOURCE を持つ 1 次元の配列になります。

SOURCE が配列の場合は、結果はランク rank(SOURCE) + 1 の配列になります。次元 DIM に沿って、結果の個々の配列エレメントは、SOURCE 内の対応する配列エレメントに等しくなります。

NCOPIES がゼロ以下の場合は、結果はゼロ・サイズの配列になります。

例

```
! A is the array (/ -4.7, 6.1, 0.3 /)

       RES = SPREAD( A, DIM = 1, NCOPIES = 3 )
! The result is   | -4.7 6.1 0.3 |
!                 | -4.7 6.1 0.3 |
!                 | -4.7 6.1 0.3 |
! DIM=1 extends each column.  Each element in RES(:,1)
! becomes a copy of A(1), each element in RES(:,2) becomes
! a copy of A(2), and so on.

       RES = SPREAD( A, DIM = 2, NCOPIES = 3 )
! The result is   | -4.7 -4.7 -4.7 |
!                 |  6.1  6.1  6.1 |
!                 |  0.3  0.3  0.3 |
! DIM=2 extends each row.  Each element in RES(1,:)
! becomes a copy of A(1), each element in RES(2,:)
! becomes a copy of A(2), and so on.

       RES = SPREAD( A, DIM = 2, NCOPIES = 0 )
! The result is (/ /) (a zero-sized array).
```

SQRT (X)

平方根です。

引き数タイプおよび属性

X タイプは実数または複素数でなければなりません。 X が複素数でない場合、値はゼロ以上でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

- X の平方根に等しい値を持ちます。
- 結果のタイプが複素数の場合は、その値はゼロ以上の実数部分を持つ主値になります。実数部分がゼロの場合は、虚数部分はゼロ以上になります。

例

SQRT (4.0) は値 2.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
SQRT	デフォルトの実数	デフォルトの実数	あり
DSQRT	倍精度実数	倍精度実数	あり
QSQRT	REAL(16)	REAL(16)	可 **1**
CSQRT **2**	デフォルトの複素数	デフォルトの複素数	あり
CDSQRT **2**	倍精度複素数	倍精度複素数	可 **1**
ZSQRT **2**	COMPLEX(8)	COMPLEX(8)	可 **1**
CQSQRT **2**	COMPLEX(16)	COMPLEX(16)	可 **1**

注:

1. IBM 拡張: 名前を引き数として渡す機能。
2. X が a + bi という形式の複素数であるとすると (ただし、i = $(-1)^{1/2}$)、 abs(X) + abs(a) は 1.797693 * 10^{308} 以下でなければなりません。

SRAND (SEED)

IBM 拡張

乱数発生ルーチン関数 RAND が使用するシード値を提供します。

引き数タイプおよび属性

SEED
スカラーでなければなりません。 **RAND** 関数に対してシード値を提供するのに使用する場合は、タイプが **REAL(4)** で、 **IRAND** サービスおよびユーティリティー関数に対してシード値を提供するのに使用する場合は、タイプが **INTEGER(4)** でなければなりません。これは、INTENT(IN) 引き数です。

クラス

サブルーチン

例

SRAND サブルーチンを使用するプログラムの例を次に示します。

```
CALL SRAND(0.5)
DO I = 1, 5
   R = RAND()
   PRINT *,R
ENDDO
END
```

上記のプログラムで生成される出力例は次のとおりです。

```
0.3984375000
0.4048461914
0.1644897461
0.1281738281E-01
0.2313232422E-01
```

IBM 拡張 の終り

SUM(ARRAY, DIM, MASK) または SUM(ARRAY, MASK)

配列内の選択されたエレメントの合計を計算します。

引き数タイプおよび属性

ARRAY エレメントを合計したい数値タイプの配列です。

DIM (オプション)
1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。

MASK (オプション)
論理式です。それが配列である場合には、形状が ARRAY に従っていなければなりません。 MASK がスカラーである場合は、その値は ARRAY 内のすべてのエレメントに適用されます。

クラス

変換関数

結果値

DIM が存在する場合は、結果は ARRAY と同じデータ型を持つ、ランク rank(ARRAY)-1 の配列になります。 DIM が脱落している場合、または MASK のランクが 1 の場合は、結果はスカラーになります。

結果は、以下のいずれかの方式で計算されます。

方式 1: ARRAY だけが指定されている場合は、結果はその配列エレメント全部の和になります。 ARRAY がゼロ・サイズ配列である場合は、結果はゼロになります。

方式 2: ARRAY と MASK が両方とも指定されている場合は、結果は値 .TRUE. を持つ MASK 内の対応する配列エレメントを持っている、 ARRAY の配列エレメントの合計となります。値が .TRUE. であるエレメントを MASK が持っていない場合は、結果はゼロになります。

方式 3: DIM も指定されている場合は、結果の値は MASK 内の対応する TRUE の配列エレメントを持つ、次元 DIM に沿った ARRAY の配列エレメントの合計になります。

Fortran 95

DIM と MASK はどちらもオプションであるため、引き数のさまざまな組み合わせが可能になります。 **-qintlog** オプションが 2 つの引き数を指定すると、2 番目の引き数は次のうちの 1 つを参照します。

- MASK。タイプが整数、論理、バイト、タイプなしの配列の場合。
- DIM。タイプが整数、バイト、またはタイプなしのスカラーの場合。
- MASK。タイプが論理のスカラーの場合。

Fortran 95 の終り

例

方式 1:

```
! Sum all the elements in an array.
      RES = SUM( (/2, 3, 4 /) )
! The result is 9 because (2+3+4) = 9
```

方式 2:

```
! A is the array (/ -3, -7, -5, 2, 3 /)
! Sum all elements that are greater than -5.
      RES = SUM( A, MASK = A .GT. -5 )
! The result is 2 because (-3 + 2 + 3) = 2
```

方式 3:

```
! B is the array | 4 2 3 |
!                | 7 8 5 |

! Sum the elements in each column.
      RES = SUM(B, DIM = 1)
! The result is | 11 10 8 | because (4 + 7) = 11
!                                   (2 + 8) = 10
!                                   (3 + 5) =  8

! Sum the elements in each row.
      RES = SUM(B, DIM = 2)
! The result is | 9 20 | because (4 + 2 + 3) =  9
!                                (7 + 8 + 5) = 20

! Sum the elements in each row, considering only
! those elements greater than two.
      RES = SUM(B, DIM = 2, MASK = B .GT. 2)
! The result is | 7 20 | because (4 + 3) =  7
!                                (7 + 8 + 5) = 20
```

SYSTEM (CMD, RESULT)

IBM 拡張

コマンドをオペレーティング・システムに渡して実行させます。コマンドが完了して、制御がオペレーティング・システムから戻るまで、現行プロセスは停止します。このサブルーチンにオプションの引き数を追加すると、オペレーティング・システムからの戻りコード情報に対して回復を実行させることができます。

引き数タイプおよび属性

CMD
: スカラーで、タイプは文字でなければなりません。これは、実行するコマンドおよび任意のコマンド行引き数を指定します。これは、INTENT(IN) 引き数です。

RESULT
: タイプが INTEGER(4) のスカラー変数でなければなりません。引き数が INTEGER(4) 変数でない場合は、コンパイラーは (S) レベルのエラー・メッセージを生成します。これは、オプションの INTENT(OUT) 引き数です。 RESULT 内に戻される情報の形式は、WAIT システム呼び出しから戻される形式と同じです。

クラス

サブルーチン

例

```
      INTEGER        ULIMIT
      CHARACTER(32)  CMD
      ...
! Check the system ulimit.
      CMD = 'ulimit > ./fort.99'
      CALL SYSTEM(CMD)
      READ(99, *) ULIMIT
      IF (ULIMIT .LT. 2097151) THEN
         ...
```

```
INTEGER RC
RC=99
CALL SYSTEM("/bin/test 1 -EQ 2",RC)
IF (IAND(RC,'ff'z) .EQ. 0) then
  RC = IAND( ISHFT(RC,-8), 'ff'z )
ELSE
  RC = -1
ENDIF
```

関連情報

下層のインプリメンテーションに関する詳細は、「*AIX Technical Reference: Base Operating System and Extensions Volume 2*」の『**system** subroutine』を参照してくださ

い。

IBM 拡張 の終り

SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)

リアルタイム・クロックから整数データを戻します。

引き数タイプおよび属性

COUNT (オプション)
INTENT(OUT) 引き数であり、スカラーで、タイプがデフォルトの整数でなければなりません。
COUNT の初期値は、プロセッサー・クロックの現行値によって異なり、 0 から COUNT_MAX までの範囲内の値になります。 COUNT は、COUNT が COUNT_MAX の値に到達するまで、各クロック・カウントごとに 1 ずつ増分します。 COUNT_MAX に到達した後、次のクロック・カウントで COUNT の値はゼロにリセットされます。

COUNT_RATE (オプション)
INTENT(OUT) 引き数であり、スカラーで、タイプがデフォルトの整数でなければなりません。デフォルトであるセンチ秒の解像度を使用すると、COUNT_RATE は 1 秒当たりのプロセッサー・クロックのカウント数になるか、またはクロックがない場合はゼロになります。

-qsclk=micro を使用してマイクロ秒の解像度を指定した場合は、 COUNT_RATE の値は 1 秒当たり 1 000 000 クロック・カウントになります。

COUNT_MAX (オプション)
INTENT(OUT) 引き数であり、スカラーで、タイプがデフォルトの整数でなければなりません。デフォルトであるセンチ秒の解像度を使用すると、COUNT_MAX は指定されたプロセッサー・クロックの最大クロック・カウント数になります。

-qsclk=micro とデフォルトの **INTEGER(4)** を使用してマイクロ秒の解像度を指定した場合、COUNT_MAX の値は 1 799 999 999 クロック・カウント、つまり約 30 分になります。

-qsclk=micro とデフォルトの **INTEGER(8)** を使用してマイクロ秒の解像度を指定した場合、COUNT_MAX の値は 8 639 999 999 クロック・カウント、つまり約 24 時間になります。

クラス

サブルーチン

例

IBM 拡張

次の例では、クロックは 24 時間クロックです。 SYSTEM_CLOCK の呼び出し後、COUNT には 1 秒当たりのクロック刻みで表された 1 日の時刻が含まれます。 1 秒当たりの刻み数は COUNT_RATE で使用できます。 COUNT_RATE 値は インプリメンテーションによって異なります。

```
INTEGER, DIMENSION(8) :: IV
TIME_SYNC: DO
CALL DATE_AND_TIME(VALUES=IV)
IHR  = IV(5)
IMIN = IV(6)
ISEC = IV(7)
CALL SYSTEM_CLOCK(COUNT=IC, COUNT_RATE=IR, COUNT_MAX=IM)
CALL DATE_AND_TIME(VALUES=IV)

IF ((IHR == IV(5)) .AND. (IMIN == IV(6)) .AND. &
  (ISEC == IV(7))) EXIT TIME_SYNC

END DO TIME_SYNC

IDAY_SEC = 3600*IHR + IMIN*60 + ISEC
IDAY_TICKS = IDAY_SEC * IR

IF (IDAY_TICKS /= IC) THEN
  STOP 'clock error'
ENDIF
END
```

IBM 拡張 の終り

関連情報

システム・クロックの解像度の指定について詳しくは、「ユーザーズ・ガイド」の **-qsclk** コンパイラー・オプションを参照してください。

TAN (X)

タンジェント (正接) 関数です。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は tan(X) に近似し、この X はラジアンの値を持ちます。

例

TAN (1.0) は値 1.5574077 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
TAN	デフォルトの実数	デフォルトの実数	あり
DTAN	倍精度実数	倍精度実数	あり
QTAN	REAL(16)	REAL(16)	可 **1**

注:

1. IBM 拡張: 名前を引き数として渡す機能。

TAND (X)

IBM 拡張

タンジェント (正接) 関数です。引き数は「度」の単位となります。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

これは tan(X) とほぼ同じ値を持ち、この X は度の値を持ちます。

例

TAND (45.0) は値 1.0 を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
TAND	デフォルトの実数	デフォルトの実数	あり
DTAND	倍精度実数	倍精度実数	あり
QTAND	REAL(16)	REAL(16)	あり

IBM 拡張 の終り

TANH (X)

双曲線正接関数を求めます。

引き数タイプおよび属性

X タイプは実数でなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

X と同じです。

結果値

結果は tanh(X) に等しい値を持ちます。

例

TANH (1.0) は値 0.76159416 (近似値) を持ちます。

特定名	引き数タイプ	結果タイプ	引き数渡し
TANH	デフォルトの実数	デフォルトの実数	あり
DTANH	倍精度実数	倍精度実数	あり
QTANH	REAL(16)	REAL(16)	可 1

注:

1. IBM 拡張: 名前を引き数として渡す機能。

TINY (X)

引き数と同じタイプおよび kind 型付きパラメーターの数を表すモデル内で最小の正の数を戻します。

引き数タイプおよび属性

X タイプは実数でなければなりません。 スカラー値または配列値を使用できます。

クラス

照会関数

結果タイプおよび属性

X と同じタイプを持つスカラー

結果値

IBM 拡張

結果は次のようになります。

`2.0`$^{\texttt{(MINEXPONENT(X)-1)}}$ `for real X`

IBM 拡張 の終り

例

IBM 拡張

TINY (X) = float(2)$^{(-126)}$ = 1.17549351e-38。 544 ページの『実データ・モデル』を参照してください。

IBM 拡張 の終り

TRANSFER (SOURCE, MOLD, SIZE)

物理表現は SOURCE と同一で、MOLD のタイプおよび型付きパラメーターで解釈された結果を戻します。

符号拡張、丸め、ブランクの埋め込み、および、他の変換方法を使用して発生させることができるその他の変更を行わずに、タイプ間で低レベルの変換を実行します。

引き数タイプおよび属性

SOURCE
ビット単位値を別のタイプに変えたいデータ・エンティティーです。これは、タイプは任意で、スカラー値でも配列値でもかまいません。

MOLD
結果として望んでいるタイプ特性を持つデータ・エンティティーです。 **MOLD** が変数の場合、値を定義する必要はありません。 これは、タイプは任意で、スカラー値でも配列値でもかまいません。その値は使用されず、そのタイプ特性だけが使用されます。

SIZE (オプション)
出力結果に対するエレメントの数です。これは、スカラー整数でなければなりません。対応する実引き数は、オプションの仮引き数であってはなりません。

クラス

変換関数

結果タイプおよび属性

MOLD と同じタイプおよび型付きパラメーター

MOLD がスカラーで、SIZE を指定しないと、結果はスカラーになります。

MOLD が配列値で、SIZE を指定しないと、結果はランク 1 の配列値になり、SOURCE を保持できるだけの物理的に十分な最小のサイズを持ちます。

SIZE を指定した場合には、結果はランク 1 で、サイズが SIZE の配列値になります。

結果値

結果の物理表現は SOURCE と同じで、結果が小さい場合は切り捨てられ、結果が大きい場合は不定の後続部分を持ちます。

物理表現は変更されないので、結果が切り捨てられない限り、TRANSFER の結果を元に戻すことができます。

```
      REAL(4) X = 3.141
      DOUBLE PRECISION I, J(6) = (/1,2,3,4,5,6/)

! Because x is transferred to a larger representation
! and then back, its value is unchanged.
      X = TRANSFER( TRANSFER( X, I ), X )

! j is transferred into a real(4) array large enough to
! hold all its elements, then back into an array of
! its original size, so its value is unchanged too.
      J = TRANSFER( TRANSFER( J, X ), J, SIZE=SIZE(J) )
```

例

IBM ▶ TRANSFER (1082130432, 0.0) は 4.0 です。◀ IBM

TRANSFER ((/1.1,2.2,3.3/), (/(0.0,0.0)/)) は長さ 2 を持つ複素数のランク 1 配列で、最初のエレメントは値 (1.1, 2.2) を持ち、2 番目のエレメントは値 3.3 を持つ実数部を持ちます。2 番目のエレメントの虚数部は不定です。

TRANSFER ((/1.1,2.2,3.3/), (/(0.0,0.0)/), 1) は、値 (/(1.1,2.2)/) を持ちます。

TRANSPOSE (MATRIX)

個々の桁を行に、個々の行を桁に変えて、2 次元配列を転置します。

引き数タイプおよび属性

MATRIX ランクが 2 のデータ型の配列です。

クラス

変換関数

結果値

結果は MATRIX と同じデータ型の 2 次元配列になります。

MATRIX の形状が (m,n) とすると、結果の形状は (n,m) になります。たとえば、MATRIX の形状が (2,3) の場合は、結果の形状は (3,2) になります。

範囲 1-n の i および範囲 1-m の j に対して、結果内の個々のエレメント (i,j) は、値 MATRIX (j,i) を持ちます。

例

```
! A is the array  | 0 -5  8 -7 |
!                 | 2  4 -1  1 |
!                 | 7  5  6 -6 |
! Transpose the columns and rows of A.
       RES = TRANSPOSE( A )
! The result is   |  0  2  7 |
!                 | -5  4  5 |
!                 |  8 -1  6 |
!                 | -7  1 -6 |
```

TRIM (STRING)

後続ブランク文字が除去された引き数を戻します。

引き数タイプおよび属性

STRING タイプは文字で、スカラーでなければなりません。

クラス

変換関数

結果タイプおよび属性

STRING と同じ kind 型付きパラメーター値を持つ文字で、その長さは STRING の長さから (STRING の後続ブランク数を引いた値) です。

結果値

- 後続ブランクが除去される以外は、結果の値は STRING と同じになります。
- STRING に非ブランク文字が含まれていない場合は、結果の長さはゼロになります。

例

TRIM ('ƀAƀBƀƀ') は値 'ƀAƀB' を持ちます。

UBOUND (ARRAY, DIM)

配列内の個々の次元の上限、または、指定された次元の上限を戻します。

引き数タイプおよび属性

ARRAY 上限を決定する配列です。その境界は定義済みでなければなりません。つまり、関連解除されているポインターや、割り振られていない割り振り可能配列であってはならず、そのサイズが想定されている場合は 1 次元だけしか検査できません。

DIM (オプション)
1 ≤ DIM ≤ rank(ARRAY) の範囲内の整数スカラーです。 対応する実引き数は、オプションの仮引き数であってはなりません。

クラス

照会関数

結果タイプおよび属性

デフォルトの整数

DIM が存在する場合は、結果はスカラーになります。 DIM が存在しない場合は、結果は ARRAY 内の個々の次元に対してエレメントを 1 つ持っている 1 次元の配列になります。

結果値

結果の中の個々のエレメントは、ARRAY の次元に対応します。 ARRAY が全体配列または配列構造体コンポーネントである場合は、これらの値は上限値に等しくなります。ARRAY が全体配列または配列構造体コンポーネントではない配列セクションまたは式である場合は、値は個々の次元のエレメント数を表す数になり、これは、元の配列の宣言された上限とは異なる場合があります。次元がゼロにサイズ決定されている場合は、結果内の対応するエレメントは、上限として宣言されている値とは無関係に、ゼロになります。

例

```
! This array illustrates the way UBOUND works with
! different ranges for dimensions.
       REAL A(1:10, -4:5, 4:-5)

       RES=UBOUND( A )
! The result is (/ 10, 5, 0 /).

       RES=UBOUND( A(:,:,:) )
! The result is (/ 10, 10, 0 /) because the argument
! is an array section.

       RES=UBOUND( A(4:10,-4:1,:) )
! The result is (/ 7, 6, 0 /), because for an array section,
! it is the number of elements that is significant.
```

UNPACK (VECTOR, MASK, FIELD)

1 次元配列からいくつかのまたは全部の配列をとり、再調整して別の (おそらく、もっと大きい) 配列にします。

引き数タイプおよび属性

VECTOR
任意のデータ型の 1 次元配列です。少なくとも MASK 内の .TRUE. 値と同数のエレメントが VECTOR 内になければなりません。

MASK
VECTOR のエレメントがアンパックされる時にどこに置かれるかを決定する論理配列です。

FIELD
マスク引き数と同じ形状と、VECTOR と同じデータ型を持っていなければなりません。そのエレメントは、対応する MASK エレメントが値 .FALSE. を持っている場合は必ず、結果配列に挿入されます。

クラス

変換関数

結果値

結果は MASK と同じ形状と、VECTOR と同じデータ型を持つ配列になります。

結果のエレメントは配列エレメント順に埋め込まれます。つまり、MASK 内の対応するエレメントが .TRUE. である場合は、結果のエレメントは VECTOR の次のエレメントで埋め込まれます。それ以外の場合は、FIELD の対応するエレメントで埋め込まれます。

例

```
! VECTOR is the array (/ 5, 6, 7, 8 /),
! MASK is | F T T |, FIELD is | -1 -4 -7 |
!         | T F F |           | -2 -5 -8 |
!         | F F T |           | -3 -6 -9 |

! Turn the one-dimensional vector into a two-dimensional
! array.  The elements of VECTOR are placed into the .TRUE.
! positions in MASK, and the remaining elements are
! made up of negative values from FIELD.
      RES = UNPACK( VECTOR, MASK, FIELD )
! The result is | -1  6  7 |
!               |  5 -5 -8 |
!               | -3 -6  8 |

! Do the same transformation, but using all zeros for the
! replacement values of FIELD.
       RES = UNPACK( VECTOR, MASK, FIELD = 0 )
! The result is | 0 6 7 |
!               | 5 0 0 |
!               | 0 0 8 |
```

VERIFY (STRING, SET, BACK)

指定された一連の文字に現れない文字ストリング内の最初の文字の位置を識別することによって、文字ストリング内のすべての文字が一連の文字に含まれていることを確認します。

引き数タイプおよび属性

STRING タイプは文字でなければなりません。

SET タイプは STRING と同じ kind 型付きパラメーターを持つ文字でなければなりません。

BACK (オプション)
タイプは論理タイプでなければなりません。

クラス

エレメント型関数

結果タイプおよび属性

デフォルトの整数

結果値

- ケース (i): BACK を指定しないか、または、値 .FALSE. を持って存在していて、SET 内にない文字が最低 1 つ STRING に含まれていると、結果の値は、 SET 内にない STRING の左端の文字位置になります。
- ケース (ii): BACK を指定していて値 .TRUE. を持ち、 SET 内にない最低 1 つの文字が STRING に含まれている場合は、結果の値は SET 内にない STRING の右端の文字位置になります。
- ケース (iii): STRING 内の個々の文字が SET 内にある場合、または STRING がゼロの長さを持っている場合は、結果の値はゼロになります。

例

- ケース (i): VERIFY ('ABBA', 'A') は値 2 を持ちます。
- ケース (ii): VERIFY ('ABBA', 'A', BACK = .TRUE.) は値 3 を持ちます。
- ケース (iii): VERIFY ('ABBA', 'AB') は値 0 を持ちます。

第 2 部 XL Fortran でのマルチスレッド・プログラミング

本節では、XL Fortran でのマルチスレッド・プログラミングの完全なリファレンスを提供します。 XL Fortran は、OpenMP Fortran API バージョン 2.0 標準仕様の完全サポートを提供するだけでなく、POSIX スレッド・ライブラリーへの Fortran インターフェースを含む pthreads ライブラリー・モジュールも提供しています。環境変数を使用した並列コードの実行の制御については、「ユーザーズ・ガイド」を参照してください。

本節には、以下の章が含まれています。

- SMP ディレクティブ
- OpenMP 実行環境およびロック・ルーチン
- Pthreads ライブラリー・モジュール

以下の部では、XL Fortran 言語のその他の特徴について説明しています。

- XL Fortran 言語
- XL Fortran 言語ユーティリティー
- ハードウェアと XL Fortran

第 13 章 SMP ディレクティブ

IBM 拡張

Symmetric Multiprocessing (SMP) ディレクティブの章には以下の節が含まれます。

- 『SMP ディレクティブの概要』
- 697 ページの『SMP ディレクティブの詳細な説明』
- 750 ページの『OpenMP ディレクティブ文節』

SMP ディレクティブの概要

この章で説明される SMP ディレクティブを使用して、並列化を制御できます。たとえば、**PARALLEL DO** ディレクティブは、ディレクティブの直後に続くループが並列で実行されることを指定します。すべての SMP ディレクティブは、注釈形式ディレクティブです。注釈形式ディレクティブの規則および構文について詳しくは、 507 ページの『注釈形式ディレクティブ』を参照してください。

- SMP ディレクティブがコンパイラーによって認識されるようにするには、**xlf_r**、**xlf_r7**、**xlf90_r**、**xlf90_r7**、または **xlf95_r**、または **xlf95_r7** 呼び出しコマンドのいずれかを使用し、**-qsmp** コンパイラー・オプションを指定してコードをコンパイルします。詳細については、この章にあるディレクティブの説明を参照してください。
- リンク・スレッド・セーフ・ライブラリーがコンパイラーによって認識されるようにするには、**xlf_r**、**xlf_r7**、**xlf90_r**、**xlf90_r7**、 **xlf95_r**、または **xlf95_r7** 呼び出しコマンドを使用してコードをコンパイルします。
- XL Fortran は、IBM での解釈による OpenMP 仕様をサポートします。コードの移植性を最大限にするために、可能な限りこれらのディレクティブを使用することをお勧めします。これらのディレクティブは、OpenMP の *trigger_constant*、 **$OMP** と一緒に使用してください。この *trigger_constant* は、他のディレクティブとは使用しないでください。

XL Fortran は、次のように分類される以下の SMP ディレクティブをサポートします。

並列領域構造体

並列構造体は、XL Fortran で OpenMP ベースの並列実行の基礎を形成します。**PARALLEL/END PARALLEL** ディレクティブの対は、基本並列構造体を形成します。実行中のスレッドが並列領域に入るたびに、そのスレッドはスレッドのチームを作成し、そのチームのマスターになります。これで、そのチームのスレッドによって、その構造体内で並列実行が行われます。並列領域には、以下のディレクティブが必要です。

PARALLEL	**END PARALLEL**

作業共用構造体

作業共用構造体は、チーム内のスレッド間の構造体によって囲まれたコードの実行を分割します。作業共用が行われるようにするには、構造体が、並列領域の動的エクステント内で囲まれていなければなりません。作業共用構造体について詳しくは、以下のディレクティブを参照してください。

DO	**END DO**
SECTIONS	**END SECTIONS**
WORKSHARE	**END WORKSHARE**

結合された並列作業共用構造体

結合された並列作業共用構造体を使用すると、単一作業共用構造体がすでに含まれている並列領域を指定できます。これらの結合構造体は、意味的に、単一作業共用構造体を囲む並列構造体を指定することと同じです。結合構造体のインプリメントについて詳しくは、以下のディレクティブを参照してください。

PARALLEL DO	**END PARALLEL DO**
PARALLEL SECTIONS	**END PARALLEL SECTIONS**
PARALLEL WORKSHARE	**END PARALLEL WORKSHARE**

同期構造体

以下のディレクティブを使用して、チーム内の複数のスレッドによる並列領域の実行を同期化することができます。

ATOMIC	
BARRIER	
CRITICAL	**END CRITICAL**
FLUSH	
ORDERED	**END ORDERED**

その他の OpenMP ディレクティブ

以下の OpenMP ディレクティブは、追加の SMP 機能を提供します。

MASTER	**END MASTER**

SINGLE	**END SINGLE**
THREADPRIVATE	

非 OpenMP SMP ディレクティブ

以下のディレクティブは、追加の SMP 機能を提供します。

DO SERIAL	
SCHEDULE	
THREADLOCAL	

SMP ディレクティブの詳細な説明

次の節は、XL Fortran でサポートされているすべての SMP ディレクティブをアルファベット順にリストしています。ディレクティブ文節については、 750 ページの『OpenMP ディレクティブ文節』を参照してください。

ATOMIC

ATOMIC ディレクティブを使用して、並列領域内の特定のメモリー位置を安全に更新することができます。 **ATOMIC** を使用する場合は、同時にメモリー位置に書き込むスレッドは 1 つだけにして、同じメモリー位置への同時書き込みによって生じることのあるエラーを避ける必要があります。

通常、共用変数が同時に複数のスレッドによって更新されている場合は、 **CRITICAL** 構造体内で共用変数を保護します。ただし、特定のプラットフォームは、変数を更新するためにアトミック・オペレーションをサポートしています。たとえば、1 つのアトミック・アクションで、メモリー位置から読み取りを行い何かを計算してその位置に書き込むハードウェア命令をサポートしているプラットフォームもあります。 **ATOMIC** ディレクティブは、可能な場合はアトミック・オペレーションを使用するようコンパイラーに指示します。このようにしないと、コンパイラーは他の機構を使用してアトミック更新を実行します。
ATOMIC ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
▶▶──ATOMIC──────────────────────────────────────────────────────────▶◀

▶▶──atomic_statement────────────────────────────────────────────────▶◀
```

atomic_statement の意味は次のとおりです。

```
▶▶──┬─update_variable──=──update_variable──operator──expression──────────┬──▶◀
    ├─update_variable──=──expression──operator──update_variable──────────┤
    ├─update_variable──=──intrinsic──(──update_variable──,──expression──)─┤
    └─update_variable──=──intrinsic──(──expression──,──update_variable──)─┘
```

update_variable
組み込みタイプのスカラー変数です。

intrinsic **max**、**min**、 **iand**、**ior** または **ieor** のいずれかです。

operator
+、**-**、*****、 **/**、**.AND.**、**.OR.**、**.EQV.**、 **.NEQV.** または **.XOR.** のいずれかです。

expression
update_variable を参照しないスカラー式です。

規則

ATOMIC ディレクティブは、直後に続くステートメントだけに適用されます。

atomic_statement の中の *expression* は、自動的には評価されません。その計算で競合状態がないようにする必要があります。

ATOMIC ディレクティブを使用し作成されたプログラム全体内の *update_variable* のストレージ・ロケーションに対するすべての参照は、同じタイプおよび同じ型付きパラメーターを持っている必要があります。

関数 *intrinsic*、演算子 *operator*、および割り当ては、組み込み関数、演算子、および割り当てでなければならず、再定義された組み込み関数、定義済み演算子または定義済み割り当てであってはなりません。

例

例 1: 次の例では、複数のスレッドがカウンターを更新しています。 **ATOMIC** は、すべての更新がカウントされるようにするために使用されています。

```
      PROGRAM P
        R = 0.0
!$OMP   PARALLEL DO SHARED(R)
        DO I=1, 10
!$OMP     ATOMIC
          R = R + 1.0
        END DO
        PRINT *,R
      END PROGRAM P
```

次のような出力になります。

```
10.0
```

例 2: 次の例では、配列 *Y* のどのエレメントがそれぞれの繰り返しで更新されるのかが確かではないので、 **ATOMIC** ディレクティブが必須です。

```
      PROGRAM P
        INTEGER, DIMENSION(10) :: Y, INDEX
        INTEGER B
        Y = 5
        READ(*,*) INDEX, B
!$OMP   PARALLEL DO SHARED(Y)
        DO I = 1, 10
!$OMP     ATOMIC
          Y(INDEX(I)) = MIN(Y(INDEX(I)),B)
        END DO
        PRINT *, Y
      END PROGRAM P
```

入力データ:

```
10 10 8 8 6 6 4 4 2 2   4
```

次のような出力になります。

```
5 4 5 4 5 4 5 4 5 4
```

例 3: 次の例は、配列を参照するために **ATOMIC** 命令を使用することはできないので無効です。

```
       PROGRAM P
       REAL ARRAY(10)
         ARRAY = 0.0
!$OMP  PARALLEL DO SHARED(ARRAY)
           DO I = 1, 10
!$OMP        ATOMIC
               ARRAY = ARRAY + 1.0
           END DO
        PRINT *, ARRAY
      END PROGRAM P
```

例 4: 次の例は無効です。 *expression* は *update_variable* を参照してはなりません。

```
   PROGRAM P
         R = 0.0
!$OMP  PARALLEL DO SHARED(R)
           DO I = 1, 10
!$OMP      ATOMIC
              R = R + R
           END DO
        PRINT *, R
      END PROGRAM P
```

関連情報

- 701 ページの『CRITICAL / END CRITICAL』
- 715 ページの『PARALLEL / END PARALLEL』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

BARRIER

BARRIER ディレクティブによって、チーム内のすべてのスレッドを同期化することができます。スレッドが **BARRIER** ディレクティブを検出すると、チーム内の他のすべてのスレッドが同じ点に達するまで待ちます。
BARRIER ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
►►──BARRIER────────────────────────────────────────────────────────►◄
```

規則

BARRIER ディレクティブは、最も近いところにある動的に対になっている **PARALLEL** ディレクティブがあれば、それをバインドします。

BARRIER ディレクティブは、**CRITICAL**、**DO** (作業共用)、**MASTER**、**PARALLEL DO**、**PARALLEL SECTIONS**、**SECTIONS**、**SINGLE**、および **WORKSHARE** ディレクティブの動的エクステント内に指定することはできません。

スレッドの 1 つが **BARRIER** ディレクティブを検出すると、チーム内のすべてのスレッドがこのディレクティブを検出しなければなりません。

すべての **BARRIER** ディレクティブと作業共用構造体は、チーム内のすべてのスレッドによって同じ順序で検出されなければなりません。

チーム内のスレッドを同期化することに加えて、**BARRIER** ディレクティブは **FLUSH** ディレクティブを暗黙指定します。

例

例 1: PARALLEL ディレクティブにバインドされる **BARRIER** ディレクティブの例です。注: ″c″ を計算するために、「a」および「b」が完全に割り当てられていることを確認する必要があるため、スレッドは待つ必要があります。

```
      SUBROUTINE SUB1
      INTEGER A(1000), B(1000), C(1000)
!$OMP PARALLEL
!$OMP DO
      DO I = 1, 1000
        A(I) = SIN(I*2.5)
      END DO
!$OMP END DO NOWAIT
!$OMP DO
      DO J = 1, 10000
        B(J) = X + COS(J*5.5)
      END DO
!$OMP END DO NOWAIT
         ...
!$OMP BARRIER
      C = A + B
!$OMP END PARALLEL
      END
```

例 2: CRITICAL セクション内で間違って使用されている **BARRIER** ディレクティブの例。 **CRITICAL** セクションに入ることができるのは、一度に 1 つのスレッドのみであるため、デッドロックになることがあります。

```
!$OMP PARALLEL DEFAULT(SHARED)

!$OMP CRITICAL
      DO I = 1, 10
        X= X + 1
!$OMP   BARRIER
        Y= Y + I*I
      END DO
!$OMP END CRITICAL
!$OMP END PARALLEL
```

関連情報

- 708 ページの『FLUSH』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 715 ページの『PARALLEL / END PARALLEL』

CRITICAL / END CRITICAL

CRITICAL 構造体では、一度に最大 1 つのスレッドによって実行されるコードの独立したブロックを定義できます。 **CRITICAL** 構造には、**CRITICAL** ディレクティブを入れ、その後にコードのブロックを続けて、最後は **END CRITICAL** ディレクティブで終了します。

CRITICAL および **END CRITICAL** ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
►►──CRITICAL──┬──────────────────┬─────────────────────────────────►◄
              └─(──lock_name──)─┘

►►──block──────────────────────────────────────────────────────────►◄

►►──END CRITICAL──┬──────────────────┬─────────────────────────────►◄
                  └─(──lock_name──)─┘
```

lock_name
: コードの各種 **CRITICAL** 構造体を区別するための名前です。

block
: 一度に最大 1 つのスレッドによって実行されるコードのブロックを表します。

規則

任意指定の *lock_name* は、グローバルに適用される名前です。同じ実行可能プログラム内の他のグローバル・エンティティーを識別するために、*lock_name* を使用することはできません。

lock_name が **CRITICAL** ディレクティブに指定されている場合、同じ *lock_name* を対応する **END CRITICAL** ディレクティブに指定しなければなりません。

同じ *lock_name* が複数の **CRITICAL** 構造体に指定されている場合、一度に 1 つのスレッドだけが **CRITICAL** 構造体のいずれか 1 つを実行することになります。複数の **CRITICAL** 構造体に異なる *lock_names* がある場合、これらの構造体は並列して実行できます。

明示的な *lock_name* のない **CRITICAL** 構造体は、すべて同じロックによって保護されます。つまり、これらの **CRITICAL** 構造体には、コンパイラーによって同じ *lock_name* が割り当てられるため、一度に 1 つのスレッドだけが無名の **CRITICAL** 構造体に入ることになります。

lock_name は、クラス 1 のローカル・エンティティーと同じ名前を共用してはなりません。

CRITICAL 構造体に分岐したり、**CRITICAL** 構造体から分岐したりすることはできません。

CRITICAL 構造体は、プログラム内のどこにでも置けます。

CRITICAL 構造体を **CRITICAL** 構造体内にネストさせることは可能ですが、デッドロック状態が発生する恐れがあります。 **-qsmp=rec_locks** コンパイラー・オプションを指定すると、デッドロックを避けることができます。詳細については、「ユーザーズ・ガイド」を参照してください。

CRITICAL および **END CRITICAL** ディレクティブは、 **FLUSH** ディレクティブを暗黙指定します。

例

例 1: この例では、**CRITICAL** 構造体が、 **PARALLEL DO** ディレクティブによってマークされた **DO** ループ内に置かれていることに注意してください。

```
      EXPR=0
!OMP$ PARALLEL DO PRIVATE (I)
      DO I = 1, 100
!OMP$   CRITICAL
          EXPR = EXPR + A(I) * I
!OMP$   END CRITICAL
      END DO
```

例 2: この例では、**CRITICAL** 構造体に *lock_name* を指定しています。

```
!SMP$ PARALLEL DO PRIVATE(T)
      DO I = 1, 100
        T = B(I) * B(I-1)
!SMP$   CRITICAL (LOCK)
          SUM = SUM + T
!SMP$   END CRITICAL (LOCK)
      END DO
```

関連情報

- 708 ページの『FLUSH』
- 161 ページの『ローカル・エンティティー』
- 715 ページの『PARALLEL / END PARALLEL』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

DO / END DO

DO (作業共用) 構造体によって、それを検出するチームのメンバー間で、ループの実行を分割することができます。 **END DO** ディレクティブによって、**DO** (作業共用) ディレクティブで指定した **DO** ループの終わりを示すことができます。
DO (作業共用) および **END DO** ディレクティブが有効なのは、**-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
        ┌──────────────────────┐
        ↓                      │
►►─DO──────┬───────────────┬───┴──────────────────────────────────►◄
           └─┬───┬─do_clause─┘
             └─,─┘

►►─do_loop────────────────────────────────────────────────────────►◄

►►──┬──────────────────────────┬──────────────────────────────────►◄
    └─END DO─┬──────────┬──────┘
             └─ NOWAIT──┘
```

do_clause の意味は次のとおりです。

```
►►──┬─firstprivate_clause─┬───────────────────────────────────────►◄
    ├─lastprivate_clause──┤
    ├─ordered_clause──────┤
    ├─private_clause──────┤
    ├─reduction_clause────┤
    └─schedule_clause─────┘
```

firstprivate_clause
: 757 ページの『FIRSTPRIVATE』を参照してください。

lastprivate_clause
: 758 ページの『LASTPRIVATE』を参照してください。

ordered_clause
: 761 ページの『ORDERED』を参照してください。

private_clause
: 762 ページの『PRIVATE』を参照してください。

reduction_clause
: 764 ページの『REDUCTION』を参照してください。

schedule_clause
: 767 ページの『SCHEDULE』を参照してください。

規則

DO (作業共用) ディレクティブに続く最初の非注釈行 (他のディレクティブは含まない) は、**DO** ループでなければなりません。この行は、無限の **DO** または **DO WHILE** ループにすることはできません。 **DO** (作業共用) ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ネストされた **DO** ループには適用されません。

END DO ディレクティブは任意指定です。 **END DO** ディレクティブを使用する場合、**DO** ループの直後に指定しなければなりません。

1 つの **DO** 構造体に、複数の **DO** ステートメントが入っていてもかまいません。 **DO** ステートメントが同じ **DO** 終了ステートメントを共用しており、 **END DO** ディレクティブが構造体に続いている場合は、作業共用 **DO** ディレクティブは、構造体の最外部の **DO** ステートメントにだけ指定できます。

END DO ディレクティブに **NOWAIT** を指定すると、早い時期にループの反復を完了させるスレッドは、ループに続く指示よりも前に実行されます。スレッドは、同じチームの他のスレッドが **DO** ループを完了するのを待ちません。 **END DO** ディレクティブに **NOWAIT** を指定しないと、各スレッドは、**DO** ループの最後にある同一のチーム内の他のスレッドをすべて待ちます。

NOWAIT 文節を指定しない場合は、 **END DO** ディレクティブによって **FLUSH** ディレクティブが暗黙指定されます。

スレッドの 1 つが **DO** (作業共用) ディレクティブを検出すると、チーム内のすべてのスレッドがこのディレクティブを検出しなければなりません。 **DO** ループのループ境界とステップ値は、チーム内のそれぞれのスレッドについて同じでなければなりません。検出されるすべての作業共用構造体と **BARRIER** ディレクティブは、チーム内のすべてのスレッドによって同じ順序で検出されなければなりません。

DO (作業共用) ディレクティブは、**CRITICAL** または **MASTER** 構造体の動的エクステント内に指定することはできません。さらに、**PARALLEL** 構造体の動的エクステント内にあるのでない限り、 **PARALLEL SECTIONS** 構造体、作業共用構造体、または **PARALLEL DO** ループの動的エクステント内のに入れることはできません。

DO (作業共用) ディレクティブの後に、別の **DO** (作業共用) ディレクティブを続けて指定することはできません。特定の **DO** ループに指定できる **DO** (作業共用) ディレクティブは 1 つだけです。

DO (作業共用) ディレクティブは、特定の **DO** ループの **INDEPENDENT** または **DO SERIAL** ディレクティブに指定することはできません。

例

例 1: PARALLEL 構造体内の、独立しているいくつかの **DO** ループの例。 **END DO** ディレクティブに **NOWAIT** が指定されているため、最初の作業同期 **DO** ループの後に、同期は実行されません。

```
!$OMP PARALLEL
!$OMP DO
      DO I = 2, N
        B(I)= (A(I) + A(I-1)) / 2.0
      END DO
!$OMP END DO NOWAIT
```

```
!$OMP DO
      DO J = 2, N
         C(J) = SQRT(REAL(J*J))
      END DO
!$OMP END DO
      C(5) = C(5) + 10
!$OMP END PARALLEL
      END
```

例 2: SHARED、および **SCHEDULE** 文節の例。

```
!$OMP PARALLEL SHARED(A)
!$OMP DO SCHEDULE(STATIC,10)
      DO I = 1, 1000
         A(I) = 1 * 4
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

例 3: 最も近い、対になっている **PARALLEL** ディレクティブにバインドする、**MASTER** および **DO** (作業共用) ディレクティブの両方の例。

```
!$OMP PARALLEL DEFAULT(PRIVATE)
      Y = 100
!$OMP MASTER
      PRINT *, Y
!$OMP END MASTER
!$OMP DO
      DO I = 1, 10
         X(I) = I
         X(I) = X(I) + Y
      END DO
!$OMP END PARALLEL
      END
```

例 4: DO (作業共用) ディレクティブの **FIRSTPRIVATE** および **LASTPRIVATE** 文節の両方の例。

```
      X = 100

!$OMP PARALLEL PRIVATE(I), SHARED(X,Y)
!$OMP DO FIRSTPRIVATE(X), LASTPRIVATE(X)
      DO I = 1, 80
         Y(I) = X + I
         X = I
      END DO
!$OMP END PARALLEL
      END
```

例 6: 共通 **DO** 終了ステートメントのあるネストされた **DO** ステートメントに適用された有効な作業共用 **DO** ディレクティブの例

```
!$OMP DO              ! A work-sharing DO directive can ONLY
                      ! precede the outermost DO statement.
      DO 100 I= 1,10
```

```
! !$OMP DO **Error**  ! Placing the OMP DO directive here is
                      ! invalid

        DO 100 J= 1,10

!          ...

100   CONTINUE
!$OMP END DO
```

関連情報

- 335 ページの『DO』
- 『DO SERIAL』
- 708 ページの『FLUSH』
- 519 ページの『INDEPENDENT』
- 369 ページの『ループの並列化』
- 712 ページの『ORDERED / END ORDERED』
- 715 ページの『PARALLEL / END PARALLEL』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 722 ページの『PARALLEL SECTIONS / END PARALLEL SECTIONS』
- 726 ページの『SCHEDULE』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

DO SERIAL

DO SERIAL ディレクティブは、ディレクティブの直後に続く **DO** ループを並列処理しないようにコンパイラーに指示します。このディレクティブは、特定の **DO** ループの自動並列化をブロックするときに便利です。 **DO SERIAL** ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
►►──DO SERIAL────────────────────────────────────────────────────────────►◄
```

規則

DO SERIAL ディレクティブに続く最初の非注釈行 (他のディレクティブは含まない) は、 **DO** ループでなければなりません。 **DO SERIAL** ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ループ内でネストされているループには適用されません。

特定の **DO** ループに指定できる **DO SERIAL** ディレクティブは 1 つだけです。 **DO SERIAL** ディレクティブは、**DO** または **PARALLEL DO** ディレクティブとともに同一の **DO** ループに指定することはできません。

DO と **SERIAL** の間のホワイト・スペースはオプションです。

このディレクティブとともに OpenMP トリガー定数を使用しないでください。

例

例 1: 内部ループ (J ループ) が並列処理されない、ネストされた DO ループの例です。

```
!$OMP PARALLEL DO PRIVATE(S,I), SHARED(A)
      DO I=1, 500
        S=0
        !SMP$ DOSERIAL
        DO J=1, 500
          S=S+1
        ENDDO
        A(I)=S+I
      ENDDO
```

例 2: ネストされたループに適用されている **DOSERIAL** ディレクティブの例です。この場合、自動並列化が使用可能になっていれば、I または K ループが並列化されることがあります。

```
      DO I=1, 100
!SMP$ DOSERIAL
        DO J=1, 100
          DO K=1, 100
            ARR(I,J,K)=I+J+K
          ENDDO
        ENDDO
      ENDDO
```

関連情報

- 703 ページの『DO / END DO』
- 335 ページの『DO』
- 369 ページの『ループの並列化』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

FLUSH

FLUSH ディレクティブを使用すると、それぞれのスレッドは他のスレッドによって生成されたデータにアクセスできるようになります。このディレクティブは、プログラムが最適化されると、コンパイラーはプロセッサー・レジスターに値を保持することがあ

るので必須です。 **FLUSH** ディレクティブを使用すると、それぞれのスレッド・ビューが一致しているということをメモリーがイメージできるようにします。
FLUSH ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

VOLATILE 属性の代わりに **FLUSH** ディレクティブを使用することによって、プログラムのパフォーマンスを向上させることができます。 **VOLATILE** 属性文節を使用すると、更新が行われた後と使用される前は常に変数がフラッシュされますが、 **FLUSH** を使用すると、変数のメモリーへの書き込みとメモリーからの読み込みは、指定した時だけ行われます。

構文

```
►►──FLUSH──┬──────────────────────────┬──────────────────────────────────────►◄
           └─(──variable_name_list──)─┘
```

規則

このディレクティブは、コード内のどこにでも指定することができます。ただし、並列領域の動的エクステントの外側で指定すると無効です。

variable_name_list を指定すると、メモリーへの書き込みとメモリーからの読み取りの対象となるのは、そのリストの中の変数だけになります (変数の書き込みまたは読み取りはまだ行われていないものと想定されます)。 *variable_name_list* の中のすべての変数は、現行有効範囲になければならず、スレッド可視でなければなりません。 スレッド可視変数は、次のいずれかになります。

- グローバル可視変数 (共通ブロックおよびモジュール・データ)
- **SAVE** 属性のあるローカル変数およびホストに関連した変数
- サブプログラム内の並列領域の中の **SHARED** 文節で指定される **SAVE** 属性のないローカル変数
- **SAVE** 属性のないアドレスが割り当てられたローカル変数
- すべてのポインターの参照解除
- 仮引き数

variable_name_list を指定しないと、すべてのスレッド可視変数がメモリーへの書き込みと読み取りの対象となります。

スレッドが **FLUSH** ディレクティブを検出すると、スレッドは、影響を受けた変数に加えられた修正をメモリーに書き込みます。スレッドは、変数のローカル・コピーがあれば (たとえば、レジスターの中に変数のコピーがある場合など)、メモリーから変数の最新のコピーを読み取ることもします。

FLUSH ディレクティブを使用することは、チーム内のすべてのスレッドにとって必須であるわけではありません。ただし、すべてのスレッド可視レッド変数が現在のものであることを保証するために、スレッド可視変数を修正するスレッドはすべて、 **FLUSH** ディレクティブを使用して、メモリー内の変数の値を更新する必要があります。**FLUSH** または **FLUSH** を暗黙指定するディレクティブの 1 つを使用しない場合は、変数の値は最新の値でないことがあります。

FLUSH はアトミックではないことに注意してください。あるディレクティブを使用して、共用ロック変数によって制御されている共用変数を **FLUSH** した後で、別のディレクティブを使用して、ロック変数を **FLUSH** する必要があります。これによって、ロック変数の前に共用変数が確実に書き込まれるようになります。

FLUSH ディレクティブが適用されるディレクティブに **NOWAIT** 文節を指定しない限り、次のディレクティブは **FLUSH** ディレクティブを暗黙指定します。

- **BARRIER**
- **CRITICAL/END CRITICAL**
- **DO/END DO**
- **END SECTIONS**
- **END SINGLE**
- **PARALLEL/END PARALLEL**
- **PARALLEL DO/END PARALLEL DO**
- **PARALLEL SECTIONS/END PARALLEL SECTIONS**
- **PARALLEL WORKSHARE/END PARALLEL WORKSHARE**
- **ORDERED/END ORDERED**

例

例 1: 次の例では、2 つのスレッドは、並列的に計算を実行し、計算が完了するときに同期化されます。

```
      PROGRAM P
      INTEGER INSYNC(0:1), IAM

!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(INSYNC)
      IAM = OMP_GET_THREAD_NUM()
      INSYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK
!$OMP FLUSH(INSYNC)
      INSYNC(IAM) = 1                         ! Each thread sets a flag
```

```
                                            ! once it has
!$OMP FLUSH(INSYNC)                         ! completed its work.
      DO WHILE (INSYNC(1-IAM) .eq. 0)       ! One thread waits for
                                            ! another to complete
!$OMP   FLUSH(INSYNC)                       ! its work.
      END DO

!$OMP END PARALLEL

      END PROGRAM P

      SUBROUTINE WORK                       ! Each thread does indep-
                                            ! endent calculations.
!       ...
!$OMP FLUSH                                 ! flush work variables
                                            ! before INSYNC
                                            ! is flushed.
      END SUBROUTINE WORK
```

例 2: 次の例は、スレッド可視ではない変数に **FLUSH** を指定しようとしているので無効です。

```
      FUNCTION F()
      INTEGER, AUTOMATIC :: i
!$OMP FLUSH(I)
      END FUNCTION F
```

MASTER / END MASTER

MASTER 構造体によって、チームのマスター・スレッドだけで実行できる、コードのブロックを定義することができます。 **MASTER** ディレクティブで始まり、それにコードのブロックが続き、最後は **END MASTER** ディレクティブで終了します。 **MASTER** および **END MASTER** ディレクティブが有効なのは、**-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
▸▸──MASTER──────────────────────────────────────────────────▸◂

▸▸──block───────────────────────────────────────────────────▸◂

▸▸──END MASTER──────────────────────────────────────────────▸◂
```

block チームのマスター・スレッドによって実行されるコードのブロックを表しています。

規則

MASTER 構造体に分岐したり、**MASTER** 構造体から分岐したりすることはできません。

MASTER ディレクティブは、最も近いところにある動的に対になっている **PARALLEL** ディレクティブがあれば、それをバインドします。

MASTER ディレクティブは、作業共用構造体の動的エクステント内、または **PARALLEL DO**、 **PARALLEL SECTIONS**、および **PARALLEL WORKSHARE** ディレクティブの動的エクステント内に指定することはできません。

MASTER 構造体に入る時、またはそこから出るときに、暗黙のバリアは存在しません。

例

例 1: PARALLEL ディレクティブにバインドされている **MASTER** ディレクティブの例です。

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP MASTER
         Y = 10.0
         X =  0.0
         DO I = 1, 4
             X = X + COS(Y) + I
         END DO
!$OMP END MASTER
!$OMP BARRIER
!$OMP DO PRIVATE(J)
         DO J = 1, 10000
              A(J) = X + SIN(J*2.5)
         END DO
!$OMP END DO
!$OMP END PARALLEL
         END
```

関連情報

- 369 ページの『ループの並列化』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 722 ページの『PARALLEL SECTIONS / END PARALLEL SECTIONS』

ORDERED / END ORDERED

ORDERED / END ORDERED ディレクティブ文節を使用すると、並列ループ内のコードのブロックの繰り返しが、ループが連続して実行される場合にループが実行する順序で実行されるようになります。構造体の外側のコードを並列的に実行したまま、**ORDERED** 構造体の内側にあるコードを予測可能な順序で実行するよう強制することができます。

ORDERED および **END ORDERED** ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
▶▶──ORDERED──────────────────────────────────────────────────▶◀

▶▶──block────────────────────────────────────────────────────▶◀

▶▶──END ORDERED──────────────────────────────────────────────▶◀
```

block 連続して実行されるコードのブロックを表しています。

規則

ORDERED ディレクティブは、 **DO** または **PARALLEL DO** ディレクティブの動的エクステントの中だけで使用することができます。 **ORDERED** 構造体に分岐したり、**ORDERED** 構造体から分岐したりすることはできません。

ORDERED ディレクティブは、最も近いところにある動的に対になっている **DO** ディレクティブまたは **PARALLEL DO** ディレクティブにバインドします。 **ORDERED** 文節は、**ORDERED** 構造体のバインド先となる **DO** ディレクティブまたは **PARALLEL DO** ディレクティブで指定する必要があります。

別の **DO** ディレクティブにバインドする **ORDERED** 構造体は、互いに独立しています。

ORDERED 構造体を一度に実行できるスレッドは 1 つだけです。スレッドは、ループが繰り返される順序で **ORDERED** 構造体に入ります。スレッドが **ORDERED** 構造体に入るのは、それまでのすべての繰り返しで構造体が実行されていた場合か、今後構造体が 1 度も実行されない場合です。

ORDERED 構造体のある並列ループのそれぞれの繰り返しで、 **ORDERED** 構造体を実行できるのは 1 度だけです。並列ループのそれぞれの繰り返しで実行できる **ORDERED** 構造は 1 つだけです。 **ORDERED** 構造体を **CRITICAL** 構造体の動的エクステント内で使用することはできません。

例

例 1: この例では、**ORDERED** 並列ループはカウントダウンされています。

```
      PROGRAM P
!$OMP PARALLEL DO ORDERED
      DO I = 3, 1, -1
!$OMP ORDERED
```

```
      PRINT *,I
!$OMP END ORDERED
      END DO
      END PROGRAM P
```

このプログラムの予期出力は、次のとおりです。

```
3
2
1
```

例 2: この例は、並列ループに **ORDERED** 構造体が 2 つあるプログラムを示しています。それぞれの繰り返しで実行できるセクションは 1 つだけです。

```
      PROGRAM P
!$OMP PARALLEL DO ORDERED
      DO I = 1, 3
        IF (MOD(I,2) == 0) THEN
!$OMP     ORDERED
            PRINT *, I*10
!$OMP     END ORDERED
        ELSE
!$OMP     ORDERED
            PRINT *, I
!$OMP     END ORDERED
        END IF
      END DO
      END PROGRAM P
```

このプログラムの予期出力は、次のとおりです。

```
1
20
3
```

例 3: この例では、プログラムは、配列のしきい値より大きいすべてのエレメントの合計数を計算します。結果が常に予測可能になるよう **ORDERED** が使用されています。四捨五入は、プログラムが実行されるたびに同じ順序で行われるので、結果は常に同じになります。

```
      PROGRAM P
        REAL :: A(1000)
        REAL :: THRESHOLD = 999.9
        REAL :: SUM = 0.0

!$OMP   PARALLEL DO ORDERED
        DO I = 1, 1000
          IF (A(I) > THRESHOLD) THEN
!$OMP       ORDERED
              SUM = SUM + A(I)
!$OMP       END ORDERED
          END IF
        END DO
      END PROGRAM P
```

注: ORDERED 文節を使用しているの時にボトルネック状態にならないようにするために、チャンク・サイズの小さい **DYNAMIC** スケジューリングまたは **STATIC** スケジューリングを使用してみることができます。詳細については、 726 ページの『SCHEDULE』を参照してください。

関連情報

- 369 ページの『ループの並列化』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 703 ページの『DO / END DO』
- 701 ページの『CRITICAL / END CRITICAL』
- 726 ページの『SCHEDULE』

PARALLEL / END PARALLEL

PARALLEL 構造体によって、スレッドのチームによって同時に実行可能なコードのブロックを定義することができます。 **PARALLEL** 構造体では、**PARALLEL** ディレクティブを使用し、それに 1 つまたは複数のコード・ブロックが続き、最後に **END PARALLEL** ディレクティブで終了します。

PARALLEL および **END PARALLEL** ディレクティブが有効なのは **-qsmp** コンパイラー・オプションが指定されている時に限られます。

構文

```
               ┌──────────────────────────┐
               ▼                          │
>>──PARALLEL───┬──────────────────────────┴──────────────────────><
               └──┬───┬──parallel_clause──┘
                  └─,─┘

>>──block────────────────────────────────────────────────────────><

>>──END PARALLEL─────────────────────────────────────────────────><
```

parallel_clause の意味は次のとおりです。

```
▶▶──┬─copyin_clause───────────┬──────────────────────────────────────────▶◀
    ├─default_clause──────────┤
    ├─firstprivate_clause─────┤
    ├─IF──(──scalar_logical_expr──)─┤
    ├─num_threads_clause──────┤
    ├─private_clause──────────┤
    ├─reduction_clause────────┤
    └─shared_clause───────────┘
```

copyin_clause
: 752 ページの『COPYIN』を参照してください。

default_clause
: 754 ページの『DEFAULT』を参照してください。

if_clause
: 756 ページの『IF』を参照してください。

firstprivate_clause
: 757 ページの『FIRSTPRIVATE』を参照してください。

num_threads_clause
: 760 ページの『NUM_THREADS』を参照してください。

private_clause
: 762 ページの『PRIVATE』を参照してください。

reduction_clause
: 764 ページの『REDUCTION』を参照してください。

shared_clause
: 769 ページの『SHARED』を参照してください。

規則

PARALLEL 構造体に分岐したり、**PARALLEL** 構造体から分岐したりすることはできません。

IF および **DEFAULT** 文節は、**PARALLEL** ディレクティブに 1 度だけ組み込めます。

対になっている **PARALLEL** 構造体の **REDUCTION** 文節に入っている変数は、対になっている **PARALLEL** 構造体内の **FIRSTPRIVATE**、**LASTPRIVATE**、または **PRIVATE** 文節に入れることはできません。さらに、内部の **PARALLEL** 構造体にある **DEFAULT** 文節を使用して **PRIVATE** にすることもできません。

並列領域で入出力操作を実行する時には注意が必要です。複数のスレッドが同じ装置で Fortran I/O ステートメントを実行する場合は、スレッドが同期化されるようにしなければなりません。そのようにしないと、動作は未定義になります。また、XL Fortran イン

プリメントでは、それぞれのスレッドは排他的に I/O 装置にアクセスしますが、OpenMP 仕様では排他的アクセスは必要ないことにも注意してください。

並列領域にバインドするディレクティブは、その並列領域に (たとえそれが逐次化されていても) バインドします。

END PARALLEL ディレクティブは、 **FLUSH** ディレクティブを暗黙指定します。

例

例 1: PARALLEL 構造体を囲む、**PRIVATE** 文節のある内部 **PARALLEL** ディレクティブの例です。注: **SHARED** 文節は、内部 **PARALLEL** 構造体に存在します。

```
!$OMP PARALLEL PRIVATE(X)
!$OMP DO
          DO I = 1, 10
            X(I) = I
!$OMP PARALLEL SHARED (X,Y)
!$OMP DO
      DO K = 1, 10
         Y(K,I)= K * X(I)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

例 2: PRIVATE、および **SHARED** 文節の両方には変数を入れることはできないことを示す例です。

```
!$OMP PARALLEL PRIVATE(A), SHARED(A)
!$OMP DO
      DO I = 1, 1000
         A(I) = I * I
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

例 3: この例は、**COPYIN** 文節の使用法を示しています。 **PARALLEL** ディレクティブによって作成されたそれぞれのスレッドには、独自の共通ブロック **BLOCK** のコピーがあります。 **COPYIN** 文節を使用すると、*FCTR* の初期値は、 **DO** ループの繰り返しを実行するスレッドの中にコピーされるようになります。

```
      PROGRAM TT
      COMMON /BLOCK/ FCTR
      INTEGER :: I, FCTR
!$OMP THREADPRIVATE(/BLOCK/)
      INTEGER :: A(100)

      FCTR = -1
      A = 0
```

```
!$OMP PARALLEL COPYIN(FCTR)
!$OMP DO
      DO I=1, 100
         FCTR = FCTR + I
         CALL SUB(A(I), I)
      ENDDO
!$OMP END PARALLEL

      PRINT *, A
      END PROGRAM

      SUBROUTINE SUB(AA, J)
      INTEGER :: FCTR, AA, J
      COMMON /BLOCK/ FCTR
!$OMP THREADPRIVATE(/BLOCK/)    ! EACH THREAD GETS ITS OWN COPY
                                ! OF BLOCK.
      AA = FCTR
      FCTR = FCTR - J
      END SUBROUTINE SUB
```

予期出力は、次のとおりです。

```
0 1 2 3 ... 96 97 98 99
```

関連情報

- 708 ページの『FLUSH』
- 『PARALLEL DO / END PARALLEL DO』
- 519 ページの『INDEPENDENT』
- 741 ページの『THREADPRIVATE』
- 703 ページの『DO / END DO』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

PARALLEL DO / END PARALLEL DO

PARALLEL DO ディレクティブでは、どのループをコンパイラーによって並列処理するかを指定できます。これは、意味上は次のものと同じです。

```
!$OMP   PARALLEL
!$OMP   DO
 ...
!$OMP ENDDO
!$OMP END PARALLEL
```

さらに、ループを並列処理するために便利な方法です。 **END PARALLEL DO** ディレクティブによって、**PARALLEL DO** ディレクティブによって指定された **DO** ループの終わりを示すことができます。

PARALLEL DO および **END PARALLEL DO** ディレクティブが有効なのは **-qsmp** コンパイラー・オプションが指定されている時に限られます。

構文

```
              ┌────────────────────────────────┐
              ▼                                │
►►──PARALLEL DO────┬──────────────────────────┬┴──────────────────────────────►◄
                   └──┬───┬──parallel_do_clause─┘
                      └─,─┘

►►──parallel_do_loop──────────────────────────────────────────────────────────►◄

►►──┬─────────────────┬───────────────────────────────────────────────────────►◄
    └─END PARALLEL DO─┘
```

parallel_do_clause の意味は次のとおりです。

```
►►──┬─copyin_clause────────────────────────────┬──────────────────────────────►◄
    ├─default_clause───────────────────────────┤
    ├─firstprivate_clause──────────────────────┤
    ├─IF──(──scalar_logical_expr──)────────────┤
    ├─lastprivate_clause───────────────────────┤
    ├─num_threads_clause───────────────────────┤
    ├─ordered_clause───────────────────────────┤
    ├─private_clause───────────────────────────┤
    ├─reduction_clause─────────────────────────┤
    ├─SCHEDULE──(──sched_type──┬─────┬──)──────┤
    │                          └─,n──┘         │
    └─shared_clause────────────────────────────┘
```

copyin_clause
: 752 ページの『COPYIN』を参照してください。

default_clause
: 754 ページの『DEFAULT』を参照してください。

if_clause
: 756 ページの『IF』を参照してください。

firstprivate_clause
: 757 ページの『FIRSTPRIVATE』を参照してください。

lastprivate_clause
: 758 ページの『LASTPRIVATE』を参照してください。

num_threads_clause
: 760 ページの『NUM_THREADS』を参照してください。

ordered_clause
: 761 ページの『ORDERED』を参照してください。

private_clause
: 762 ページの『PRIVATE』を参照してください。

reduction_clause
: 764 ページの『REDUCTION』を参照してください。

schedule_clause
: 767 ページの『SCHEDULE』を参照してください。

shared_clause
: 769 ページの『SHARED』を参照してください。

規則

PARALLEL DO ディレクティブに続く最初の非注釈行は、**DO** ループでなければなりません。この行は、無限の **DO** または **DO WHILE** ループにすることはできません。 **PARALLEL DO** ディレクティブは、ディレクティブの直後にある **DO** ループにのみ適用され、ネストされた **DO** ループには適用されません。

DO ループを **PARALLEL DO** ディレクティブによって指定する場合、**END PARALLEL DO** ディレクティブは任意指定になります。 **END PARALLEL DO** ディレクティブを使用する場合、**DO** ループの直後に指定しなければなりません。

1 つの **DO** 構造体に、複数の **DO** ステートメントが入っていてもかまいません。 **DO** ステートメントが同じ **DO** 終了ステートメントを共用しており、 **END PARALLEL DO** ディレクティブが構造体に続いている場合は、 **PARALLEL DO** ディレクティブは、構造体の最外部の **DO** ステートメントにだけ指定できます。

PARALLEL DO ディレクティブの後に、 **DO** (作業共用) または **DO SERIAL** ディレクティブを続けることはできません。 1 つの **DO** ループに指定できる **PARALLEL DO** ディレクティブは 1 つだけです。

検出されるすべての作業共用構造体と **BARRIER** ディレクティブは、チーム内のすべてのスレッドによって同じ順序で検出されなければなりません。

1 つの **DO** ループについて、**PARALLEL DO** ディレクティブと **INDEPENDENT** ディレクティブを併用することはできません。

注: INDEPENDENT ディレクティブを使うと、複数の **HPF** 処理系でコードを共有できます。複数ベンダー間の移植性を最大限に確保するには、 **PARALLEL DO** ディレクティブを使用しなければなりません。 **PARALLEL DO** ディレクティブは規定ディレクティブですが、**INDEPENDENT** はループの特性に関する断定ディレクティブです。 **INDEPENDENT** ディレクティブの詳細については、519 ページを参照してください。

IF 文節は、**PARALLEL DO** ディレクティブに 1 度だけ組み込めます。

IF 式は、並列構造体のコンテキストの外側で評価されます。 **IF** 式の中の関数参照は、副次作用を与えるものであってはなりません。

デフォルトでは、ネストされた並列ループは、**IF** 文節の設定にかかわりなく逐次化されます。このデフォルトは、**-qsmp=nested_par** コンパイラー・オプションを使用して変更できます。

内部の **DO** ループの **REDUCTION** 変数が、外側の **DO** ループの **PRIVATE** または **LASTPRIVATE** 文節に入れる場合、その変数は内部の **DO** ループの前で初期化しなければなりません。

外側の **DO** ループの **INDEPENDENT** ディレクティブの **REDUCTION** 文節にある変数は、**PRIVATE** または **LASTPRIVATE** 文節の *data_scope_entity_list* にも入れることはできません。

並列領域で入出力操作を実行する時には注意が必要です。複数のスレッドが同じ装置で Fortran I/O ステートメントを実行する場合は、スレッドが同期化されるようにしなければなりません。そのようにしないと、動作は未定義になります。また、XL Fortran インプリメントでは、それぞれのスレッドは排他的に I/O 装置にアクセスしますが、OpenMP 仕様では排他的アクセスは必要ないことにも注意してください。

並列領域にバインドするディレクティブは、その並列領域に (たとえそれが逐次化されていても) バインドします。

例

例 1: LASTPRIVATE 文節の有効例

```
!$OMP PARALLEL DO PRIVATE(I), LASTPRIVATE (X)
      DO I = 1,10
        X = I * I
        A(I) = X * B(I)
      END DO
      PRINT *, X                       ! X has the value 100
```

例 2: REDUCTION 文節の有効例

```
!$OMP PARALLEL DO PRIVATE(I), REDUCTION(+:MYSUM)
      DO I = 1, 10
        MYSUM = MYSUM + IARR(I)
      END DO
```

例 3: SHARED とマークされていて、複数のスレッドからアクセスされる変数を CRITICAL 構造体以外では使用できないようにする有効例。

```
!$OMP  PARALLEL DO SHARED (X)
     DO I = 1, 10
             A(I) = A(I) * I
```

```
!$OMP       CRITICAL
              X = X + A(I)
!$OMP       END CRITICAL
          END DO
```

例 4: END PARALLEL DO ディレクティブの有効例

```
         REAL A(100), B(2:100), C(100)
!$OMP PARALLEL DO
         DO I = 2, 100
             B(I) = (A(I) + A(I-1))/2.0
         END DO
!$OMP END PARALLEL DO
!$OMP PARALLEL DO
         DO  J = 1, 100
            C(J) = X + COS(J*5.5)
         END DO
!$OMP END PARALLEL DO
         END
```

関連情報

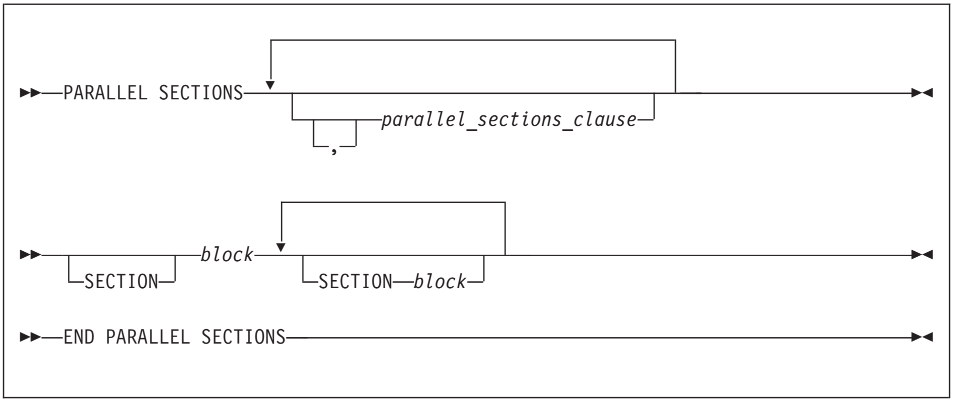
- 369 ページの『ループの並列化』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』
- 335 ページの『DO』
- 703 ページの『DO / END DO』
- 519 ページの『INDEPENDENT』
- 369 ページの『ループの並列化』
- 712 ページの『ORDERED / END ORDERED』
- 715 ページの『PARALLEL / END PARALLEL』
- 『PARALLEL SECTIONS / END PARALLEL SECTIONS』
- 726 ページの『SCHEDULE』
- 741 ページの『THREADPRIVATE』

PARALLEL SECTIONS / END PARALLEL SECTIONS

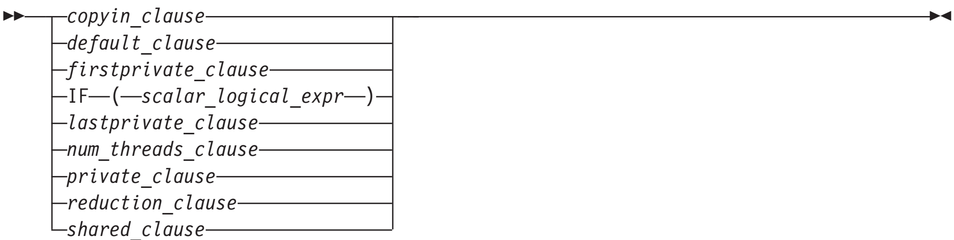
PARALLEL SECTIONS 構造体では、コンパイラーが同時に実行できるコードの個々のブロックを定義できます。 **PARALLEL SECTIONS** 構造体では、**PARALLEL SECTIONS** ディレクティブを使用し、その後に 1 つまたは複数のコード・ブロック (**SECTION** ディレクティブによって区切られている) を続け、最後に **END PARALLEL SECTIONS** ディレクティブで終了します。

PARALLEL SECTIONS、**SECTION**、および **END PARALLEL SECTIONS** ディレクティブが有効なのは、**-qsmp** コンパイラー・オプションが指定されている時に限られます。

構文



parallel_sections_clause の意味は次のとおりです。



copyin_clause
: 752 ページの『COPYIN』を参照してください。

default_clause
: 754 ページの『DEFAULT』を参照してください。

firstprivate_clause
: 757 ページの『FIRSTPRIVATE』を参照してください。

if_clause
: 756 ページの『IF』を参照してください。

lastprivate_clause
: 758 ページの『LASTPRIVATE』を参照してください。

num_threads_clause
: 760 ページの『NUM_THREADS』を参照してください。

private_clause
: 762 ページの『PRIVATE』を参照してください。

reduction_clause
764 ページの『REDUCTION』を参照してください。

shared_clause
769 ページの『SHARED』を参照してください。

規則

PARALLEL SECTIONS 構造体には、上記の構文で示されているように、区切りディレクティブと、区切りディレクティブで囲まれているコード・ブロックが含まれます。以下の規則は、セクション にも当てはまります。セクションとは、区切りディレクティブ内にあるコード・ブロックのことです。

SECTION ディレクティブは、コード・ブロックの始まりをマークします。少なくとも 1 つの **SECTION** とそのコード・ブロックを **PARALLEL SECTIONS** 構造体に入れなければなりません。ただし、**SECTION** ディレクティブは、最初のセクションには指定する必要がありません。ブロックの最後は、別の **SECTION** ディレクティブか、または **END PARALLEL SECTIONS** ディレクティブによって区切られます。

PARALLEL SECTIONS 構造体は、指定のコード・セクションの並列実行を指定するために使用できます。セクションの実行順序には前提事項はありません。セクションが他のセクションを妨害することはありません。ただし、 **CRITICAL** 構造体内で生じる妨害は例外です。 **CRITICAL** 構造体の外で発生する妨害の定義については、 519ページを参照してください。

PARALLEL SECTIONS 構造体によって定義されているコード・ブロックに分岐したり、そのコード・ブロックから分岐することはできません。

コンパイラーは、並列で実行されるスレッドの数とセクションの数という演算項目の数に基づいて、スレッド間で作業を分割する方法を決定します。したがって、1 つのスレッドが **SECTION** を複数回実行したり、 **SECTION** をまったく実行しないこともあります。

検出されるすべての作業共用構造体と **BARRIER** ディレクティブは、チーム内のすべてのスレッドによって同じ順序で検出されなければなりません。

PARALLEL SECTIONS 構造体の場合、**PRIVATE** 文節に入っていない変数は、デフォルトで **SHARED** として想定されます。

PARALLEL SECTIONS 構造体の場合、外側の **DO** ループの **INDEPENDENT** ディレクティブまたは **PARALLEL DO** ディレクティブの **REDUCTION** 文節にある変数は、**PRIVATE** 文節の *data_scope_entity_list* にも入れることはできません。

内部 **PARALLEL SECTIONS** 構造体の **REDUCTION** 変数が、外側の **DO** ループまたは **PARALLEL SECTIONS** 構造体の **PRIVATE** 文節に入れる場合、その変数は内部の **PARALLEL SECTIONS** 構造体の前で初期化しなければなりません。

PARALLEL SECTIONS 構造体は、**CRITICAL** 構造体に入れることはできません。

並列領域で入出力操作を実行する時には注意が必要です。複数のスレッドが同じ装置で Fortran I/O ステートメントを実行する場合は、スレッドが同期化されるようにしなければなりません。そのようにしないと、動作は未定義になります。また、XL Fortran インプリメントでは、それぞれのスレッドは排他的に I/O 装置にアクセスしますが、OpenMP 仕様では排他的アクセスは必要ないことにも注意してください。

並列領域にバインドするディレクティブは、その並列領域に (たとえそれが逐次化されていても) バインドします。

END PARALLEL SECTIONS ディレクティブは、 **FLUSH** ディレクティブを暗黙指定します。

例

例 1:

```
!$OMP PARALLEL SECTIONS
!$OMP   SECTION
          DO I = 1, 10
            C(I) = MAX(A(I),A(I+1))
          END DO
!$OMP   SECTION
          W = U + V
          Z = X + Y
!$OMP END PARALLEL SECTIONS
```

例 2: この例では、指標変数 I が **PRIVATE** として宣言されています。最初の任意指定の **SECTION** ディレクティブが省略されていることに注意してください。

```
!$OMP PARALLEL SECTIONS PRIVATE(I)
          DO I = 1, 100
            A(I) = A(I) * I
          END DO
!$OMP   SECTION
          CALL NORMALIZE (B)
          DO I = 1, 100
            B(I) = B(I) + 1.0
          END DO
!$OMP   SECTION
          DO I = 1, 100
            C(I) = C(I) * C(I)
          END DO
!$OMP END PARALLEL SECTIONS
```

例 3: 複数セクションにまたがって変数 C にデータの依存性があるため、この例は無効です。

```
!$OMP PARALLEL SECTIONS
!$OMP   SECTION
          DO I = 1, 10
            C(I) = C(I) * I
```

```
          END DO
!$OMP   SECTION
          DO K = 1, 10
            D(K) = C(K) + K
          END DO
!$OMP END PARALLEL SECTIONS
```

関連情報

- 715 ページの『PARALLEL / END PARALLEL』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 519 ページの『INDEPENDENT』
- 741 ページの『THREADPRIVATE』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

PARALLEL WORKSHARE / END PARALLEL WORKSHARE

PARALLEL WORKSHARE 構造体は、 **WORKSHARE** ディレクティブを **PARALLEL** 構造体の内部に組み込むための簡易書式メソッドを提供します。

構文

```
                    ┌──────────────────────────────────┐
                    ▼                                  │
►►──PARALLEL WORKSHARE──┬────────────────────────────────┬┴──────────►◄
                        └─┬───┬─parallel_workshare_clause─┘
                          └─,─┘

►►──block─────────────────────────────────────────────────────────────►◄

►►──END PARALLEL WORKSHARE────────────────────────────────────────────►◄
```

ここで *parallel_workshare_clause* は、 **PARALLEL** または **WORKSHARE** ディレクティブのどちらかによって受け入れられる、任意のディレクティブの文節です。

関連情報

- 715 ページの『PARALLEL / END PARALLEL』
- 747 ページの『WORKSHARE』

SCHEDULE

SCHEDULE ディレクティブでは、並列化のためのチャンク方式を指定できます。スケジューリング・タイプまたはチャンク・サイズに応じて、異なる方法で作業がスレッドに割り当てられます。

SCHEDULE ディレクティブは、**-qsmp** オプション・コンパイラー・オプションが指定された場合にのみ有効です。

構文

```
▶▶—SCHEDULE—(—sched_type—┬──────┬—)———————————————————————————▶◀
                          └─,—n──┘
```

n 　*n* は、正の宣言式でなければなりません。*sched_type* **RUNTIME** には、*n* を指定してはなりません。

sched_type
　AFFINITY、**DYNAMIC**、**GUIDED**、**RUNTIME**、または **STATIC** です。

sched_type パラメーターについて詳しくは、**SCHEDULE** 文節を参照してください。

number_of_iterations
　並列化するループの繰り返しの回数。

number_of_threads
　プログラムによって使用されるスレッドの数。

規則

SCHEDULE ディレクティブは、有効範囲単位の仕様の部分に入れなければなりません。

1 つの **SCHEDULE** ディレクティブだけを有効範囲単位の仕様の部分に入れることができます。

SCHEDULE ディレクティブは次のものに適用されます。

- 明示的なスケジューリング・タイプがまだ指定されていない有効範囲単位のすべてのループ。各ループには、**PARALLEL DO** ディレクティブの **SCHEDULE** 文節を使用してスケジューリング・タイプを指定できます。
- コンパイラーによって生成され、自動並列化処理によって並列化したループ。たとえば、**SCHEDULE** ディレクティブは、 **FORALL**、**WHERE**、I/O 暗黙 **DO**、および配列コンストラクター暗黙 **DO** のために生成されたループに適用されます。

チャンク・サイズ *n* のために宣言式に入れる仮引き数、またはそこで参照される仮引き数は、**SUBROUTINE** または **FUNCTION** ステートメント、および指定のサブプログラムにあるすべての **ENTRY** ステートメントにも入れなければなりません。

指定したチャンク・サイズ *n* が繰り返しの数より大きい場合、ループは並列化されずに、単一スレッドで実行されます。

チャンク化のアルゴリズムを複数指定する場合、コンパイラーは次の優先順位に従います。

1. **SCHEDULE** 文節から **PARALLEL DO** ディレクティブ。
2. **SCHEDULE** ディレクティブ。
3. **-qsmp** コンパイラー・オプションの **schedule** サブオプション。「ユーザーズ・ガイド」の『**-qsmp** オプション』を参照してください。
4. **XLSMPOPTS** 実行時オプション。「ユーザーズ・ガイド」の『**XLSMPOPTS**』を参照してください。
5. 実行時のデフォルト値 (つまり **STATIC**)。

例

例 1: 条件は次のとおりです。

```
number of iterations = 1000
number of threads = 4
```

GUIDED スケジューリング・タイプを使用。この場合のチャンク・サイズは次のとおりです。

```
250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1
```

繰り返しは次のチャンクに分割されます。

```
chunk  1 = iterations    1 to  250
chunk  2 = iterations  251 to  438
chunk  3 = iterations  439 to  579
chunk  4 = iterations  580 to  685
chunk  5 = iterations  686 to  764
chunk  6 = iterations  765 to  823
chunk  7 = iterations  824 to  868
chunk  8 = iterations  869 to  901
chunk  9 = iterations  902 to  926
chunk 10 = iterations  927 to  945
chunk 11 = iterations  946 to  959
chunk 12 = iterations  960 to  970
chunk 13 = iterations  971 to  978
chunk 14 = iterations  979 to  984
chunk 15 = iterations  985 to  988
chunk 16 = iterations  989 to  991
chunk 17 = iterations  992 to  994
chunk 18 = iterations  995 to  996
chunk 19 = iterations  997 to  997
chunk 20 = iterations  998 to  998
chunk 21 = iterations  999 to  999
chunk 22 = iterations 1000 to 1000
```

作業分割の一例は次のとおりです。

```
thread 1 executes chunks 1 5 10 13 18 20
thread 2 executes chunks 2 7  9 14 16 22
thread 3 executes chunks 3 6 12 15 19
thread 4 executes chunks 4 8 11 17 21
```

例 2: 条件は次のとおりです。

```
number of iterations = 100
number of threads = 4
```

AFFINITY スケジューリング・タイプを使用。この場合、繰り返しは次の区画に分割されます。

```
partition 1 = iterations  1 to 25
partition 2 = iterations 26 to  50
partition 3 = iterations 51 to  75
partition 4 = iterations 76 to 100
```

区画は次のチャンクに分割されます。

```
chunk 1a = iterations   1 to  13
chunk 1b = iterations  14 to  19
chunk 1c = iterations  20 to  22
chunk 1d = iterations  23 to  24
chunk 1e = iterations  25 to  25

chunk 2a = iterations  26 to  38
chunk 2b = iterations  39 to  44
chunk 2c = iterations  45 to  47
chunk 2d = iterations  48 to  49
chunk 2e = iterations  50 to  50

chunk 3a = iterations  51 to  63
chunk 3b = iterations  64 to  69
chunk 3c = iterations  70 to  72
chunk 3d = iterations  73 to  74
chunk 3e = iterations  75 to  75

chunk 4a = iterations  76 to  88
chunk 4b = iterations  89 to  94
chunk 4c = iterations  95 to  97
chunk 4d = iterations  98 to  99
chunk 4e = iterations 100 to 100
```

作業分割の一例は次のとおりです。

```
thread 1 executes chunks 1a 1b 1c 1d 1e 4d
thread 2 executes chunks 2a 2b 2c 2d
thread 3 executes chunks 3a 3b 3c 3d 3e 2e
thread 4 executes chunks 4a 4b 4c 4e
```

このシナリオでは、スレッド 1 がその区画内のチャンクをすべて実行し終え、スレッド 4 の区画から使用可能なチャンクを取っています。同様に、スレッド 3 はその区画内のチャンクをすべて実行し終え、スレッド 2 の区画から使用可能なチャンクを取りました。

例 3: 条件は次のとおりです。

```
number of iterations = 1000
number of threads = 4
```

DYNAMIC スケジューリング・タイプを使用。この場合のチャンク・サイズは次のとおりです。

```
100 100 100 100 100 100 100 100 100 100
```

繰り返しは次のチャンクに分割されます。

```
chunk  1 = iterations   1 to  100
chunk  2 = iterations 101 to  200
chunk  3 = iterations 201 to  300
chunk  4 = iterations 301 to  400
chunk  5 = iterations 401 to  500
chunk  6 = iterations 501 to  600
chunk  7 = iterations 601 to  700
chunk  8 = iterations 701 to  800
chunk  9 = iterations 801 to  900
chunk 10 = iterations 901 to 1000
```

作業分割の一例は次のとおりです。

```
thread 1 executes chunks 1  5  9
thread 2 executes chunks 2  8
thread 3 executes chunks 3  6  10
thread 4 executes chunks 4  7
```

例 4: 条件は次のとおりです。

```
number of iterations = 100
number of threads = 4
```

STATIC スケジューリング・タイプを使用。繰り返しは次のチャンクに分割されます。

```
chunk 1 = iterations  1 to  25
chunk 2 = iterations 26 to  50
chunk 3 = iterations 51 to  75
chunk 4 = iterations 76 to 100
```

作業分割の一例は次のとおりです。

```
thread 1 executes chunks 1
thread 2 executes chunks 2
thread 3 executes chunks 3
thread 4 executes chunks 4
```

関連情報

- 335 ページの『DO』

SECTIONS / END SECTIONS

SECTIONS 構造体は、チーム内のスレッドが並列で実行するコードの別個のブロックを定義しています。

SECTIONS および **END SECTIONS** ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
           ┌─────────────────────────────┐
           ▼                             │
►►─SECTIONS──┬─────────────────────────┬─┴──────────────────────►◄
             └─┬───┬─sections_clause─┘
               └─,─┘

                             ┌──────────────────────┐
                             ▼                      │
►►─┬─────────┬─block───┬────────────────────┬─┴───────────────►◄
   └─SECTION─┘         └─SECTION──block─┘

►►─END SECTIONS─┬────────┬──────────────────────────────────►◄
                └─NOWAIT─┘
```

sections_clause の意味は次のとおりです。

```
►►──┬─firstprivate_clause─┬──────────────────────────────────►◄
    ├─lastprivate_clause──┤
    ├─private_clause──────┤
    └─reduction_clause────┘
```

firstprivate_clause
: 757 ページの『FIRSTPRIVATE』を参照してください。

lastprivate_clause
: 758 ページの『LASTPRIVATE』を参照してください。

private_clause
: 762 ページの『PRIVATE』を参照してください。

reduction_clause
: 764 ページの『REDUCTION』を参照してください。

規則

SECTIONS 構造体は、チーム内のすべてのスレッドによって検出されるか、チーム内のどのスレッドによっても検出されないかのどちらかでなければなりません。検出されるすべての作業共用構造体と **BARRIER** ディレクティブは、チーム内のすべてのスレッドによって同じ順序で検出されなければなりません。

SECTIONS 構造体には、上記の構文で示されているように、区切りディレクティブと、区切りディレクティブで囲まれているコード・ブロックが含まれます。構造体の中には、コードのブロックが最低 1 つ入っていなければなりません。

SECTION ディレクティブは、最初のコード・ブロックを除いたすべてのコード・ブロックの先頭に指定する必要があります。ブロックの最後は、別の **SECTION** ディレクティブか、または **END SECTIONS** ディレクティブによって区切られます。

SECTIONS 構造体によって囲まれているコード・ブロックに分岐したり、そのコード・ブロックから分岐することはできません。すべての **SECTION** ディレクティブは、**SECTIONS/END SECTIONS** ディレクティブ対の字句エクステントの中になければなりません。

コンパイラーは、並列で実行されるチーム内のスレッドの数とセクションの数という演算項目の数に基づいて、スレッド間で作業を分割する方法を決定します。したがって、1 つのスレッドが **SECTION** を複数回実行することがあります。また、**SECTION** がチーム内のスレッドによってまったく実行されないこともあります。

ディレクティブが並列で実行されるようにするには、 **SECTIONS/END SECTIONS** 対を並列領域の動的エクステント内に置く必要があります。このようにしないと、ブロックは逐次で実行されます。

SECTIONS ディレクティブに **NOWAIT** を指定する場合、早い時期にループの反復を完了させるスレッドは、 **SECTIONS** 構造体の後ろにある命令よりも先に実行されます。 **NOWAIT** 文節を指定しないと、それぞれのスレッドは、同じチーム内の他のすべてのスレッドが **END SECTIONS** ディレクティブに達するのを待ちます。 しかし、**SECTIONS** 構造体の最初には、暗黙指定された **BARRIER** はありません。

SECTIONS ディレクティブは、 **CRITICAL** または **MASTER** ディレクティブの動的エクステント内に指定することはできません。

同じ **PARALLEL** ディレクティブにバインドする **SECTIONS**、 **DO** または **SINGLE** ディレクティブをネストさせることはできません。

BARRIER および **MASTER** ディレクティブを、 **SECTIONS** ディレクティブの動的エクステントの中に入れることはできません。

END SECTIONS ディレクティブは、 **FLUSH** ディレクティブを暗黙指定します。

例

例 1: この例は、**PARALLEL** 領域内での **SECTIONS** 構造体の有効な使用法を示しています。

```
      INTEGER :: I, B(500), S, SUM
! ...
      S = 0
      SUM = 0
!$OMP PARALLEL SHARED(SUM), FIRSTPRIVATE(S)
!$OMP SECTIONS REDUCTION(+: SUM), LASTPRIVATE(I)
!$OMP SECTION
        S = FCT1(B(1::2))  ! Array B is not altered in FCT1.
```

```
        SUM = SUM + S
! ...
!$OMP SECTION
        S = FCT2(B(2::2))  ! Array B is not altered in FCT2.
        SUM = SUM + S
! ...
!$OMP SECTION
        DO I = 1, 500      ! The local copy of S is initialized
          S = S + B(I)     ! to zero.
        END DO
        SUM = SUM + S
! ...
!$OMP END SECTIONS
! ...
!$OMP DO REDUCTION(-: SUM)
      DO J=I-1, 1, -1    ! The loop starts at 500 -- the last
                         ! value from the previous loop.
        SUM = SUM - B(J)
      END DO

!$OMP MASTER
      SUM = SUM - FCT1(B(1::2)) - FCT2(B(2::2))
!$OMP END MASTER
!$OMP END PARALLEL
! ...
                   ! Upon termination of the PARALLEL
                   ! region, the value of SUM remains zero.
```

例 2: この例は、ネストされた **SECTIONS** の有効な使用法を示しています。

```
!$OMP PARALLEL
!$OMP MASTER
      CALL RANDOM_NUMBER(CX)
      CALL RANDOM_NUMBER(CY)
      CALL RANDOM_NUMBER(CZ)
!$OMP END MASTER

!$OMP SECTIONS
!$OMP SECTION
!$OMP   PARALLEL
!$OMP   SECTIONS PRIVATE(I)
!$OMP   SECTION
          DO I=1, 5000
            X(I) = X(I) + CX
          END DO
!$OMP   SECTION
          DO I=1, 5000
            Y(I) = Y(I) + CY
          END DO
!$OMP   END SECTIONS
!$OMP   END PARALLEL

!$OMP SECTION
!$OMP   PARALLEL SHARED(CZ,Z)
!$OMP   DO
```

```
        DO I=1, 5000
          Z(I) = Z(I) + CZ
        END DO
!$OMP   END DO
!$OMP   END PARALLEL
!$OMP END SECTIONS NOWAIT

! The following computations do not
! depend on the results from the
! previous section.

!$OMP DO
      DO I=1, 5000
        T(I) = T(I) * CT
      END DO
!$OMP END DO
!$OMP END PARALLEL
```

関連情報

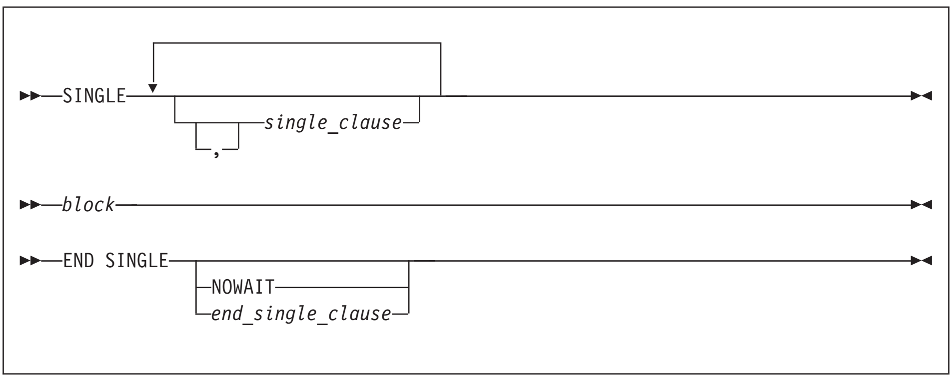
- 715 ページの『PARALLEL / END PARALLEL』
- 700 ページの『BARRIER』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 519 ページの『INDEPENDENT』
- 741 ページの『THREADPRIVATE』
- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

SINGLE / END SINGLE

SINGLE / END SINGLE ディレクティブ構造体を使用して、チーム内の 1 つのスレッドだけに実行される囲まれたコードを指定することができます。

SINGLE ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文



single_clause の意味は次のとおりです。

```
>>─┬─private_clause──────┬────────────────────────────────────────><
   └─firstprivate_clause─┘
```

private_clause
762 ページの『PRIVATE』を参照してください。

firstprivate_clause
757 ページの『FIRSTPRIVATE』を参照してください。

end_single_clause の意味は次のとおりです。

```
   ┌──────────────────────────┐
   ▼                          │
>>───copyprivate_clause──┴─,─┴───────────────────────────────────><
```

NOWAIT

copyprivate_clause
.

規則

SINGLE 構造体の内部で囲まれているブロックに分岐したり、そのブロックから分岐したりすることはできません。

SINGLE 構造体は、チーム内のすべてのスレッドによって検出されるか、チーム内のどのスレッドによっても検出されないかのどちらかでなければなりません。検出されるすべての作業共用構造体と **BARRIER** ディレクティブは、チーム内のすべてのスレッドによって同じ順序で検出されなければなりません。

END SINGLE ディレクティブに **NOWAIT** を指定すると、 **SINGLE** 構造体を実行していないスレッドは、 **SINGLE** 構造体の後ろにある命令よりも先に実行されます。 **NOWAIT** 文節を指定しないと、それぞれのスレッドは、構造体を実行しているスレッドが **END SINGLE** ディレクティブに達するまで、 **END SINGLE** ディレクティブで待ちます。 同じ **END SINGLE** ディレクティブの中で、 **NOWAIT** および **COPYPRIVATE** を一緒に指定できません。

SINGLE 構造体の最初には、暗黙指定された **BARRIER** はありません。 **NOWAIT** 文節を指定しないと、 **END SINGLE** ディレクティブで **BARRIER** ディレクティブが暗黙指定されます。

同じ **PARALLEL** ディレクティブにバインドする **SECTIONS**、 **DO** および **SINGLE** ディレクティブをネストさせることはできません。

SINGLE ディレクティブを、 **CRITICAL** および **MASTER** ディレクティブの動的エクステントの中に入れることはできません。 **BARRIER** および **MASTER** ディレクティブを、 **SINGLE** ディレクティブの動的エクステントの中に入れることはできません。

SINGLE 構造体を囲んでいる **PARALLEL** 構造体の中で、変数を **PRIVATE**、 **FIRSTPRIVATE**、 **LASTPRIVATE** または **REDUCTION** として指定してある場合は、同じ変数を **SINGLE** 構造体の **PRIVATE** または **FIRSTPRIVATE** 文節の中で指定することはできません。

SINGLE ディレクティブは、最も近いところにある動的に対になっている **PARALLEL** ディレクティブがあれば、それをバインドします。

例

例 1: この例では、**SINGLE** 構造体に入る前にすべてのスレッドが作業を終えるよう、 **BARRIER** ディレクティブが使用されています。

```
      REAL :: X(100), Y(50)
!     ...
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)

!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE

      CALL WORK(Y)
!$OMP END PARALLEL
```

例 2: この例では、**SINGLE** 構造体があるので、コード・ブロックを実行するスレッドは 1 つだけになります。このケースでは、配列 *B* は **DO** (作業共用) 構造体の中で初期化されています。初期化後に、合計を出すために 1 つのスレッドが使用されています。

```
      INTEGER :: I, J
      REAL :: B(500,500), SM
!     ...

      J = ...
      SM = 0.0
!$OMP PARALLEL
!$OMP DO PRIVATE(I)
      DO I=1, 500
        CALL INITARR(B(I,:), I)       ! initialize the array B
      ENDDO
!$OMP END DO

!$OMP SINGLE                          ! employ only one thread
      DO I=1, 500
        SM = SM + SUM(B(J:J+1,I))
      ENDDO
```

```
!$OMP END SINGLE

!$OMP DO PRIVATE(I)
      DO I=500, 1, -1
        CALL INITARR(B(I,:), 501-I)   ! re-initialize the array B
      ENDDO
!$OMP END PARALLEL
```

例 3: この例は、**PRIVATE** 文節の有効な使用法を示しています。配列 *X* は、**SINGLE** 構造体に対して **PRIVATE** になっています。構造体のすぐ後に続く配列 *X* を参照する場合、これは未定義になります。

```
      REAL :: X(2000), A(1000), B(1000)

!$OMP PARALLEL
!     ...
!$OMP SINGLE PRIVATE(X)
      CALL READ_IN_DATA(X)
      A = X(1::2)
      B = X(2::2)
!$OMP END SINGLE
!     ...
!$OMP END PARALLEL
```

例 4: この例では、*TMP* を割り振る際に **LASTPRIVATE** 変数 *I* が使用されており、**SINGLE** 構造体の中で **PRIVATE** 変数が使用されています。

```
      SUBROUTINE ADD(A, UPPERBOUND)
        INTEGER :: A(UPPERBOUND), I, UPPERBOUND
        INTEGER, ALLOCATABLE :: TMP(:)
!  ...
!$OMP   PARALLEL
!$OMP   DO LASTPRIVATE(I)
        DO I=1, UPPERBOUND
          A(I) = I + 1
        ENDDO
!$OMP   END DO

!$OMP   SINGLE FIRSTPRIVATE(I), PRIVATE(TMP)
        ALLOCATE(TMP(0:I-1))
        TMP = (/ (A(J),J=I,1,-1) /)
!  ...
        DEALLOCATE(TMP)
!$OMP   END SINGLE
!$OMP   END PARALLEL
!  ...
      END SUBROUTINE ADD
```

例 5: この例では、変数 *I* の値をユーザーが入力します。次に、この値は、**END SINGLE** ディレクティブの **COPYPRIVATE** 文節を使用して、チーム内の他のすべてのスレッドで、対応する変数 *I* にコピーされます。

```
        INTEGER I
  !$OMP PARALLEL PRIVATE (I)
  !     ...
```

```
!$OMP SINGLE
      READ (*, *) I
!$OMP END SINGLE COPYPRIVATE (I)   ! In all threads in the team, I
                                   ! is equal to the value
!     ...                          ! that you entered.
!$OMP END PARALLEL
```

例 6: この例では、**POINTER** 属性を持つ変数 *J* が、 **END SINGLE** ディレクティブ上の **COPYPRIVATE** 文節に指定されています。 *J* の値 (それが指しているオブジェクトの値ではありません) は、同じチーム内の他のすべてのスレッドで、対応する変数 *J* にコピーされます。オブジェクト自体は、チーム内のすべてのスレッド間で共用されます。

```
      INTEGER, POINTER :: J
!$OMP PARALLEL PRIVATE (J)
! ...
!$OMP SINGLE
      ALLOCATE (J)
      READ (*, *) J
!$OMP END SINGLE COPYPRIVATE (J)
!$OMP ATOMIC
      J = J + OMP_GET_THREAD_NUM()
!$OMP BARRIER
!$OMP SINGLE
      WRITE (*, *) 'J = ', J   ! The result is the sum of all values added to
                               ! J. This result shows that the pointer object
                               ! is shared by all threads in the team.
      DEALLOCATE (J)
!$OMP END SINGLE
!$OMP END PARALLEL
```

関連情報

- 700 ページの『BARRIER』
- 701 ページの『CRITICAL / END CRITICAL』
- 708 ページの『FLUSH』
- 711 ページの『MASTER / END MASTER』
- 715 ページの『PARALLEL / END PARALLEL』

THREADLOCAL

THREADLOCAL ディレクティブは、スレッド固有の共通データを宣言するときに使用します。これは、**COMMON** ブロック内のデータへのアクセスを逐次化する方法にもなります。

このディレクティブを利用するために **-qsmp** コンパイラー・オプションを利用する必要はありませんが、必要なライブラリーにリンクするための呼び出しコマンドは **xlf_r**、**xlf_r7**、**xlf90_r**、**xlf90_r7**、 **xlf95_r**、または **xlf95_r7** でなければなりません。

構文

```
                             ┌─,─────────────────────────┐
                             ▼                           │
►►─THREADLOCAL──┬────┬──/─common_block_name─/─┴───────────────────────────►◄
                └─::─┘
```

規則

名前付きブロックだけを **THREADLOCAL** で宣言できます。通常、名前付き共通ブロックに適用される規則および制約はすべて **THREADLOCAL** で宣言されている共通ブロックに適用されます。名前付き共通ブロックに適用される規則と制約の詳細については、 314 ページの『COMMON』を参照してください。

THREADLOCAL ディレクティブは、有効範囲単位の *specification_part* に入れなければなりません。共通ブロックを **THREADLOCAL** ディレクティブに入れる場合、同じ有効範囲単位の **COMMON** ステートメントにもこの共通ブロックを宣言しなければなりません。 **THREADLOCAL** ディレクティブは、**COMMON** ステートメントの前にも後にも入れることができます。有効範囲単位の *specification_part* の詳細については、 181 ページの『メインプログラム』を参照してください。

共通ブロックが **PURE** サブプログラムに宣言されている場合、共通ブロックに **THREADLOCAL** 属性を指定することはできません。

THREADLOCAL 共通ブロックのメンバーを **NAMELIST** ステートメントに入れることはできません。

使用関連付けがある共通ブロックは、**USE** ステートメントを含む有効範囲単位内の **THREADLOCAL** として宣言することはできません。

THREADLOCAL 共通ブロック内で宣言されているポインターは、 **-qinit=f90ptr** コンパイラー・オプションの影響を受けません。

THREADLOCAL 共通ブロック内にあるオブジェクトは、並列ループと並列セクションで使用できます。しかし、これらのオブジェクトはループの繰り返し間、および並列セクションのコード・ブロック間で暗黙的に共用されます。つまり、有効範囲単位内では、すべてのアクセス可能共通ブロック (**THREADLOCAL** として宣言されていてもいなくても) は、その有効範囲単位の並列ループと並列セクションに **SHARED** 属性があるということです。

共通ブロックが有効範囲単位内で **THREADLOCAL** として宣言される場合、その共通ブロックを宣言または参照しており、直接または間接的に有効範囲単位によって参照されているサブプログラムは、その有効範囲単位を実行している同一のスレッドによって実行しなければなりません。共通ブロックを宣言する 2 つのプロシージャーが異なるスレッドによって実行されると、共通ブロックに **THREADLOCAL** が宣言されるなら、

それらのプロシージャーは異なる共通ブロックのコピーを得ることになります。スレッドは、以下のいずれかの方法で作成できます。
- 明示的に *pthreads* ライブラリー呼び出しを介して行う。
- 暗黙的にコンパイラーを使用して並列ループを実行して行う。
- 暗黙的にコンパイラーを使用して並列セクションを実行して行う。

共通ブロックを 1 つの有効範囲単位内の **THREADLOCAL** として宣言する場合、その共通ブロックは、共通ブロックを宣言する各有効範囲単位ごとに **THREADLOCAL** として宣言しなければなりません。

SAVE 属性がない **THREADLOCAL** 共通ブロックをサブプログラム内で宣言する場合、ブロックのメンバーはサブプログラムの RETURN または END で未定義になります。ただし、直接または間接にサブプログラムを参照している共通ブロックにアクセスできる有効範囲単位がほかに 1 つ以上ある場合はこの限りではありません。

THREADLOCAL ディレクティブと **THREADPRIVATE** ディレクティブの両方に、同じ *common_block_name* を指定することはできません。

例 1: 次のプロシージャー ″FORT_SUB″ は、2 つのスレッドによって呼び出されます。

```
SUBROUTINE FORT_SUB(IARG)
  INTEGER IARG

  CALL LIBRARY_ROUTINE1()
  CALL LIBRARY_ROUTINE2()
  ...
END SUBROUTINE FORT_SUB
SUBROUTINE LIBRARY_ROUTINE1()
  COMMON /BLOCK/ R                ! The SAVE attribute is required for the
  SAVE /BLOCK/                    ! common block because the program requires
                                  ! that the block remain defined after
  !IBM* THREADLOCAL /BLOCK/       ! library_routine1 is invoked.
    R = 1.0
    ...
END SUBROUTINE LIBRARY_ROUTINE1
SUBROUTINE LIBRARY_ROUTINE2()
  COMMON /BLOCK/ R
  SAVE /BLOCK/
  !IBM* THREADLOCAL /BLOCK/

  ... = R
  ...
END SUBROUTINE LIBRARY_ROUTINE2
```

例 2: ″FORT_SUB″ が複数のスレッドによって呼び出されます。 ″FORT_SUB″ と ″ANOTHER_SUB″ は両方とも /BLOCK/ を THREADLOCAL として宣言するため、これは無効な例です。これらは、共通ブロックを共有しようとしますが、異なるスレッドによって実行されます。

```
SUBROUTINE FORT_SUB()
  COMMON /BLOCK/ J
  INTEGER :: J
  !IBM* THREADLOCAL /BLOCK/        ! Each thread executing FORT_SUB
                                   ! obtains its own copy of /BLOCK/
  INTEGER A(10)

  ...
  !IBM* INDEPENDENT
  DO INDEX = 1,10
    CALL ANOTHER_SUB(A(I))
  END DO
  ...

END SUBROUTINE FORT_SUB
SUBROUTINE ANOTHER_SUB(AA)         ! Multiple threads
are used to execute ANOTHER_SUB
  INTEGER AA
  COMMON /BLOCK/ J                 ! Each thread obtains a new copy of the
  INTEGER :: J                     !  common block /BLOCK/
  !IBM* THREADLOCAL /BLOCK/
  ...
  AA = J                           ! The value of 'J' is undefined.
END SUBROUTINE ANOTHER_SUB
```

関連情報

- 「ユーザーズ・ガイド」の『**-qdirective**』
- 「ユーザーズ・ガイド」の『**-qinit**』
- 314 ページの『COMMON』
- 181 ページの『メインプログラム』
- **/usr/lpp/xlf/samples/modules/threadlocal** ディレクトリーには、threadlocal を使用する方法と C でスレッドを作成する方法を示した 1 つ以上のサンプル・プログラムがあります。

THREADPRIVATE

THREADPRIVATE ディレクティブを使用すると、あるスレッドに対してはプライベートとして、ただしそのスレッド内ではグローバルとして、名前付き共通ブロックまたは名前付き変数を指定できます。共通ブロックまたは変数 **THREADPRIVATE** を宣言すると、チーム内の各スレッドは、共通ブロックまたは変数の別個のコピーを維持します。**THREADPRIVATE** 共通ブロックまたは変数に書き込まれたデータは、そのスレッドに対してプライベートのままであり、チーム内の他のスレッドに対して可視ではありません。

プログラムの逐次セクションと **MASTER** セクションでは、マスター・スレッドが持つ名前付き共通ブロックと変数のコピーだけがアクセス可能になります。

PARALLEL、**PARALLEL DO**、 **PARALLEL SECTIONS** または **PARALLEL WORKSHARE** ディレクティブで **COPYIN** 文節を使用して、並列領域に入るときに、

マスター・スレッドが持つ名前付き共通ブロックまたは名前付き変数のコピーの中のデータが、それぞれのスレッドが持つ共通ブロックまたは変数のプライベート・コピーにコピーされるよう指定します。

THREADPRIVATE ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
►►—THREADPRIVATE—(—threadprivate_entity_list—)———————————————————►◄
```

threadprivate_entity_list の意味は次のとおりです。

```
►►——┬variable_name———————————┬——————————————————————————————————►◄
    └/ common_block_name /———┘
```

common_block_name
: スレッドに対してプライベートにされる共通ブロックの名前です。

variable_name
: スレッドに対してプライベートにされる変数の名前です。

規則

THREADPRIVATE 変数または共通ブロックを指定したり、 **PRIVATE**、**FIRSTPRIVATE**、**LASTPRIVATE**、 **SHARED**、または **REDUCTION** 文節の中でこの共通ブロックを構成する変数を指定することはできません。

THREADPRIVATE 変数は、**SAVE** 属性を持つ必要があります。モジュールの有効範囲内に宣言された変数または共通ブロックの場合は、 **SAVE** 属性が暗黙のうちに示されています。モジュールの有効範囲外の変数を宣言する場合は、 **SAVE** 属性を指定する必要があります。

THREADPRIVATE ディレクティブの中で指定できるのは、名前付き変数および名前付き共通ブロックだけです。

変数は、変数が宣言されている有効範囲内の **THREADPRIVATE** ディレクティブでのみ使用でき、 **THREADPRIVATE** 変数または共通ブロックは、特定の有効範囲内で一度だけ指定することができます。変数は、共通ブロックのエレメントであったり、**EQUIVALENCE** ステートメントに宣言したりすることはできません。

THREADPRIVATE ディレクティブと **THREADLOCAL** ディレクティブの両方に、同じ *common_block_name* を指定することはできません。

名前付き共通ブロックに適用されるすべての規則と制約も、 **THREADPRIVATE** で宣言されている共通ブロックに適用されます。 314 ページの『COMMON』を参照してください。

ある有効範囲単位内で共通ブロックを **THREADPRIVATE** として宣言すると、その共通ブロックが宣言されている他のすべての有効範囲単位内でも、この共通ブロックを **THREADPRIVATE** として宣言しなければなりません。

並列領域に入るときには、**THREADPRIVATE** 変数、または、 **THREADPRIVATE** 共通ブロック内の変数は、**COPYIN** 文節内に宣言されるとき、次の基準に従います。

- 変数に **POINTER** 属性があり、マスター・スレッドの変数のコピーがターゲットに関連している場合は、その変数の各コピーは、同じターゲットに関連します。マスター・スレッドのポインターが関連解除される場合は、その変数の各コピーも関連解除されます。変数のスレッドのコピーに未定義の関連付け状況がある場合は、その変数の各コピーにも、未定義の関連付け状況があります。
- **POINTER** 属性のない変数の各コピーには、その変数のマスター・スレッドのコピーと同じ値が割り当てられます。

プログラムの最初の並列領域に入る際は、 **COPYIN** 文節に指定されていない、**THREADPRIVATE** 変数または **THREADPRIVATE** 共通ブロック内の変数は、以下の基準に従います。

- 変数に **ALLOCATABLE** 属性がある場合は、その変数の各コピーの初期の割り振り状況は、現在割り振られていない状況になります。
- 変数に **POINTER** 属性がある場合は、そのポインターは、明示的またはデフォルトの初期化のいずれかを通して関連解除されます。その変数の各コピーの関連付け状況は、関連解除になります。そうでない場合は、ポインターの関連付け状況は未定義になります。
- 変数に **ALLOCATABLE** または **POINTER** 属性のどちらもない場合で、明示的またはデフォルトの初期化のどちらかを通して定義された場合は、その変数の各コピーは定義済みになり。変数が未定義の場合、その変数の各コピーも未定義になります。

プログラムの後続の並列領域に入る際は、 **COPYIN** 文節に指定されていない、**THREADPRIVATE** 変数または **THREADPRIVATE** 共通ブロック内の変数は、以下の基準に従います。

- **OMP_DYNAMIC** 環境変数、または動的スレッドを使用可能にするために **omp_set_dynamic** サブルーチンを使用している場合は、その変数のスレッドのコピーの定義および関連付け状況は未定義になり、割り振り状況も未定義になります。
- 動的スレッドが使用不可の場合で、変数のスレッドのコピーが定義済みの場合は、定義、関連付け、または割り振りの状況および定義は保持されます。

共通ブロックの名前には、使用関連付けまたはホスト関連付けによってアクセスすることはできません。したがって、**THREADPRIVATE** ディレクティブを構成している有効

範囲単位内で共通ブロックが宣言されている場合、名前付き共通ブロックを使用できるのは、 **THREADPRIVATE** ディレクティブの中だけです。ただし、共通ブロックの変数には、使用関連付けまたはホスト関連付けによってアクセスすることができます。詳細については、 164 ページの『ホスト関連付け』と 166 ページの『使用関連付け』を参照してください。

-qinit=f90ptr コンパイラー・オプションは、 **THREADPRIVATE** 共通ブロックの中で宣言したポインターに影響を与えることはありません。

DEFAULT 文節は、**THREADPRIVATE** 共通ブロックの中の変数に影響を与えることはありません。

例

例 1: この例では、**PARALLEL DO** ディレクティブは、 **SUB1** を呼び出す複数のスレッドを呼び出しています。 **SUB1** の中の共通ブロック **BLK** は、 **SUB1** によって呼び出されるサブルーチン **SUB2** を持つスレッドに固有のデータを共用しています。

```
      PROGRAM TT
        INTEGER :: I, B(50)

!$OMP   PARALLEL DO SCHEDULE(STATIC, 10)
        DO I=1, 50
          CALL SUB1(I, B(I))       ! Multiple threads call SUB1.
        ENDDO
      END PROGRAM TT

      SUBROUTINE SUB1(J, X)
        INTEGER :: J, X, A(100)
        COMMON /BLK/ A
!$OMP   THREADPRIVATE(/BLK/)   ! Array a is private to each thread.
!  ...
        CALL SUB2(J)
        X = A(J) + A(J + 50)
!  ...
      END SUBROUTINE SUB1

      SUBROUTINE SUB2(K)
        INTEGER :: C(100)
        COMMON /BLK/ C
!$OMP   THREADPRIVATE(/BLK/)
!  ...
        C = K
!  ...                         ! Since each thread has its own copy of
                               ! common block BLK, the assignment of
                               ! array C has no effect on the copies of
                               ! that block owned by other threads.
END SUBROUTINE SUB2
```

例 2: この例では、それぞれのスレッドは並列セクションの中で、共通ブロック **ARR** の独自のコピーを持っています。あるスレッドが共通ブロック変数 **TEMP** を初期化する場合、初期値は他のスレッドに可視ではありません。

```
      PROGRAM ABC
        INTEGER :: I, TEMP(100), ARR1(50), ARR2(50)
        COMMON /ARR/ TEMP
!$OMP   THREADPRIVATE(/ARR/)
        INTERFACE
          SUBROUTINE SUBS(X)
            INTEGER :: X(:)
          END SUBROUTINE
        END INTERFACE
! ...
!$OMP   PARALLEL SECTIONS
!$OMP   SECTION                   ! The thread has its own copy of the
! ...                             ! common block ARR.
          TEMP(1:100:2) = -1
          TEMP(2:100:2) = 2
          CALL SUBS(ARR1)
! ...
!$OMP   SECTION                   ! The thread has its own copy of the
! ...                             ! common block ARR.
          TEMP(1:100:2) = 1
          TEMP(2:100:2) = -2
          CALL SUBS(ARR2)
! ...
!$OMP    END PARALLEL SECTIONS
! ...
        PRINT *, SUM(ARR1), SUM(ARR2)
      END PROGRAM ABC

      SUBROUTINE SUBS(X)
        INTEGER :: K, X(:), TEMP(100)
        COMMON /ARR/ TEMP
!$OMP   THREADPRIVATE(/ARR/)
!  ...
        DO K = 1, UBOUND(X, 1)
          X(K) = TEMP(K) + TEMP(K + 1)    ! The thread is accessing its
                                          ! own copy of
                                          ! the common block.
        ENDDO
! ...
      END SUBROUTINE SUBS
```

このプログラムの予期出力は、次のとおりです。

```
50 -50
```

例 3: 次の例では、共通ブロックの外側にあるローカル変数は、 **THREADPRIVATE** で宣言されます。

```
      MODULE MDL
        INTEGER          :: A(2)
        INTEGER, POINTER :: P
        INTEGER, TARGET  :: T
!$OMP THREADPRIVATE(A, P)
      END MODULE MDL
```

```
      PROGRAM MVAR
      USE MDL

      INTEGER :: I
      INTEGER OMP_GET_THREAD_NUM

      CALL OMP_SET_NUM_THREADS(2)
      A = (/1, 2/)
      T = 4
      P => T

!$OMP PARALLEL PRIVATE(I) COPYIN(A, P)
      I = OMP_GET_THREAD_NUM()
      IF (I .EQ. 0) THEN
        A(1) = 100
        T = 5
      ELSE IF (I .EQ. 1) THEN
        A(2) = 200
      END IF
!$OMP END PARALLEL

!$OMP PARALLEL PRIVATE(I)
      I = OMP_GET_THREAD_NUM()
      IF (I .EQ. 0) THEN
        PRINT *, 'A(2) = ', A(2)
      ELSE IF (I .EQ. 1) THEN
        PRINT *, 'A(1) = ', A(1)
        PRINT *, 'P => ', P
      END IF
!$OMP END PARALLEL

      END PROGRAM MVAR
```

動的スレッドのメカニズムが使用不可の場合は、予期される出力は次のようになります。

```
A(2) = 2
A(1) = 1
P => 5
または
A(1) = 1
P => 5
A(2) = 2
```

関連情報

- 314 ページの『COMMON』
- 「ユーザーズ・ガイド」の **OMP_DYNAMIC** 環境変数
- 781 ページの『omp_set_dynamic』
- 715 ページの『PARALLEL / END PARALLEL』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 722 ページの『PARALLEL SECTIONS / END PARALLEL SECTIONS』

WORKSHARE

WORKSHARE ディレクティブを使用すると、配列操作を並行して実行できます。**WORKSHARE** ディレクティブは、囲まれたコード・ブロックに関連したタスクを、*作業単位* に分割します。スレッドのチームが **WORKSHARE** ディレクティブを検出すると、チーム内のスレッドはタスクを共用し、各*作業単位* を正確に一度実行します。

WORKSHARE ディレクティブが有効なのは、 **-qsmp** コンパイラー・オプションが指定されているときに限られます。

構文

```
►►──WORKSHARE──────────────────────────────────────────────────►◄

►►──block──────────────────────────────────────────────────────►◄

►►──END WORKSHARE──┬────────┬──────────────────────────────────►◄
                   └─NOWAIT─┘
```

block ステートメントの構造化ブロック。これは、**WORKSHARE** 構造体の字句エクステント内での作業共用を可能にします。ステートメントの実行は同期化されるので、その結果が別のステートメントに従属しているステートメントは、結果が要求される前に評価されます。*block* には、以下のいずれかを含むことができます。

- 配列割り当てステートメント
- **ATOMIC** ディレクティブ
- **CRITICAL** 構造体
- **FORALL** 構造体
- **FORALL** ステートメント
- **PARALLEL** 構造体
- **PARALLEL DO** 構造体
- **PARALLEL SECTION** 構造体
- **PARALLEL WORKSHARE** 構造体
- スカラー割り当てステートメント
- **WHERE** 構造体
- **WHERE** ステートメント

配列操作の一部として使用できる変形可能組み込み関数は、以下のとおりです。

ALL	**MATMUL**	**PRODUCT**
ANY	**MAXLOC**	**RESHAPE**
COUNT	**MAXVAL**	**SPREAD**
CSHIFT	**MINLOC**	**SUM**
DOT_PRODUCT	**MINVAL**	**TRANSPOSE**
EOSHIFT	**PACK**	**UNPACK**

block には、字句的に囲まれた **PARALLEL** 構造体にバインドされたステートメントを含むこともできます。これらのステートメントには制限はありません。

block 内のユーザー定義の関数呼び出しはいずれも、エレメント型である必要があります。

WORKSHARE ディレクティブ内に囲まれたステートメントは、*作業単位* に分割されます。*作業単位* の定義は、評価されるステートメントによって異なります。*作業単位* は、以下のように定義されます。

- **配列式:** 配列式の各エレメントの評価が、1 つの*作業単位* になります。上記にリストされた変形可能組み込み関数はいずれも、いくつかの*作業単位* に分割されます。
- **割り当てステートメント:** 配列割り当てステートメントでは、配列内の各エレメントの割り当てが 1 つの*作業単位* になります。スカラー割り当てステートメントの場合、割り当て操作が 1 つの*作業単位* になります。
- **構造体:** 各 **CRITICAL** 構造体の評価が 1 つの*作業単位* になります。**WORKSHARE** 構造体内に含まれた各 **PARALLEL** 構造体が 1 つの*作業単位* になります。スレッドの新規チームは、囲まれた **PARALLEL** 構造体の字句エクステント内に含まれる、ステートメントを実行します。**FORALL** 構造体またはステートメントでは、マスク式の評価、反復スペースの指定内で発生する式、およびマスクされた割り当てが、*作業単位* になります。**WHERE** 構造体またはステートメントでは、マスク式およびマスクされた割り当ての評価が*作業単位* になります。
- **ディレクティブ:** 1 つの **ATOMIC** ディレクティブおよびその割り当てに対する各スカラー変数の更新が 1 つの*作業単位* になります。
- **ELEMENTAL 関数: ELEMENTAL** 関数に対する引き数が配列の場合は、配列の各エレメントに対する関数の適用が 1 つの*作業単位* になります。

上記の定義のうち、どれもブロック内のステートメントに適用されない場合は、そのステートメントが 1 つの*作業単位* になります。

規則

WORKSHARE 構造体内のステートメントを確実に並行で実行するために、構造体は、並列領域の動的エクステント内に囲まなければなりません。並列領域の動的エクステントの外側で **WORKSHARE** 構造体を見つけたスレッドは、その構造体内でステートメントを逐次評価します。

WORKSHARE ディレクティブは、最も近いところにある動的に囲まれた **PARALLEL** ディレクティブがあれば、それをバインドします。

同じ **PARALLEL** ディレクティブとバインドする、 **DO**、**SECTIONS**、**SINGLE** および **WORKSHARE** ディレクティブをネストできません。

CRITICAL、**MASTER**、または **ORDERED** ディレクティブの動的エクステント内には、 **WORKSHARE** ディレクティブを指定できません。

WORKSHARE 構造体の動的エクステント内には、 **BARRIER**、**MASTER**、または **ORDERED** ディレクティブを指定できません。

配列割り当て、スカラー割り当て、マスクされた配列割り当て、または *block* 内のプライベート変数に割り当てる **FORALL** 割り当ての場合は、結果は未定義です。

block 内の配列式が値を参照する場合は、プライベート変数の関連付け状況または割り振り状況は、各スレッドが同じ値を計算するまで、式の値は未定義です。

NO WAIT 文節を **WORKSHARE** 構造体の最後で指定しないと、 **BARRIER** ディレクティブが暗黙的に指定されます。

WORKSHARE 構造体は、チーム内のすべてのスレッドで見つかるか、まったくないかのどちらかである必要があります。

例

例 1: 次の例では、**WORKSHARE** ディレクティブが、マスクされた式を並列に評価します。

```
!$OMP WORKSHARE
      FORALL (I = 1 : N, AA(1, I) = 0) AA(1, I) = I
      BB = TRANSPOSE(AA)
      CC = MATMUL(AA, BB)
!$OMP ATOMIC
      S = S + SUM(CC)
!$OMP END WORKSHARE
```

例 2: 次の例には、**WORKSHARE** 構造体の一部として、ユーザー定義の **ELEMENTAL** が含まれます。

```
!$OMP WORKSHARE
      WHERE (AA(1, :) /= 0.0) AA(1, :) = 1 / AA(1, :)
      DD = TRANS(AA(1, :))
```

```
!$OMP END WORKSHARE
      ELEMENTAL REAL FUNCTION TRANS(ELM) RESULT(RES)
      REAL, INTENT(IN) :: ELM
      RES = ELM * ELM + 4
      END FUNCTION
```

関連情報

- 697 ページの『ATOMIC』
- 700 ページの『BARRIER』
- 701 ページの『CRITICAL / END CRITICAL』
- 726 ページの『PARALLEL WORKSHARE / END PARALLEL WORKSHARE』
- 「ユーザーズ・ガイド」の『**-qsmp** オプション』

OpenMP ディレクティブ文節

以下の OpenMP ディレクティブ文節を使用すると、並列構造体内の変数の有効範囲属性を指定できます。また、この節の **IF**、**NUM_THREADS**、**ORDERED**、および **SCHEDULE** 文節を使用して、並列領域の並列環境を制御することもできます。詳しくは、詳細なディレクティブの説明を参照してください。

COPYIN	**FIRSTPRIVATE**	**PRIVATE**
COPYPRIVATE	**LASTPRIVATE**	**REDUCTION**
DEFAULT	**NUM_THREADS**	**SCHEDULE**
IF	**ORDERED**	**SHARED**

ディレクティブ文節のグローバル規則

変数名または共通ブロック名を文節に複数回指定してはなりません。

共通ブロックのメンバーである変数、共通ブロック名、または変数名は、次の例外を除いて、同じディレクティブの複数の文節に指定してはなりません。

- 名前付き共通ブロックまたは名前付き変数は、同じディレクティブの **FIRSTPRIVATE** および **LASTPRIVATE** として定義できます。
- **NUM_THREADS** 文節に指定されている変数は、同じディレクティブの別の文節に指定することができます。
- **IF** 文節に指定されている変数は、同じディレクティブの別の文節に指定することができます。

変数の有効範囲を変更する文節を指定しない場合は、ディレクティブの影響を受ける変数のデフォルト有効範囲は、**SHARED** になります。

並列領域の動的エクステント内で参照されるプロシージャーの中で宣言された、**SAVE** または **STATIC** 属性を持つローカル変数には、暗黙の **SHARED** 属性があります。 並

列領域の動的エクステント内で参照されるプロシージャーの中で宣言された、**SAVE** または **STATIC** 属性を持たないローカル変数には、暗黙の **PRIVATE** 属性があります。

並列領域の動的エクステント内で参照されるプロシージャーの中で宣言された、モジュールの共通ブロックおよび変数のメンバーには、それらが **THREADLOCAL** または **THREADPRIVATE** 共通ブロックおよびモジュール変数である場合を除いて、暗黙の **SHARED** 属性があります。

以下のような状況では、並列または作業共用構造体が実行されている間、ディレクティブの **PRIVATE**、**FIRSTPRIVATE**、**LASTPRIVATE**、または **REDUCTION** 文節で使用される変数あるいは変数サブオブジェクトについて、参照、定義、定義解除、関連付け状況の変更、割り振り状況の変更、または実引き数としての指定を行ってはなりません。

- ディレクティブ構造体がある有効範囲単位以外の有効範囲単位の中
- 変数形式設定式の中

変数を **PRIVATE**、**FIRSTPRIVATE**、**LASTPRIVATE**、または **REDUCTION** として宣言することができます。この変数がすでに他の変数とストレージが関連している場合であってもそうです。ストレージ関連付けが、**EQUIVALENCE** ステートメントまたは **COMMON** ブロックの中で宣言されている変数について存在することがあります。**PRIVATE**、**FIRSTPRIVATE**、**LASTPRIVATE**、または **REDUCTION** 変数とストレージが関連している変数の場合は、次のようになります。

- **PRIVATE**、**FIRSTPRIVATE**、 **LASTPRIVATE** または **REDUCTION** 変数とストレージが関連した変数の内容、割り振り状況、および関連付け状況は、並列構造体に入る時は未定義です。
- 関連した変数の割り振り状況、関連付け状況、および内容は、 **PRIVATE**、**FIRSTPRIVATE**、 **LASTPRIVATE** または **REDUCTION** 変数を定義する場合、またはその変数の割り振り状況または関連付け状況を定義する場合は、未定義になります。
- **PRIVATE**、**FIRSTPRIVATE**、 **LASTPRIVATE** または **REDUCTION** 変数の割り振り状況、関連付け状況、および内容は、関連した変数を定義する場合、または関連した変数の割り振り状況または関連付け状況を定義する場合は、未定義になります。

ポインターおよび OpenMP Fortran API バージョン 2.0

OpenMP Fortran API バージョン 2.0 は、**PRIVATE** 文節の変数または変数サブオブジェクトが **POINTER** 属性を持つことができるようにします。ポインターの関連付け状況は、スレッドが作成されたときと、スレッドが破棄されたときに未定義になります。

IBM 拡張

XL Fortran は、**FIRSTPRIVATE** または **LASTPRIVATE** 文節の変数または変数サブオブジェクトが **POINTER** 属性を持つことができるようにする拡張機能を提供します。ス

レッド作成時の **FIRSTPRIVATE** ポインターの場合、ポインターの各コピーがオリジナルと同じ関連付け状況を受け取ります。ポインターが **LASTPRIVATE** 文節で使用される場合、ポインターは、最後の繰り返しまたは **SECTION** の終わりの関連付け状況を保持します。

IBM 拡張 の終り

OpenMP Fortran API バージョン 2.0 標準との完全な準拠を維持するには、**POINTER** 変数は、**PRIVATE** 文節にのみ適用してください。

COPYIN

COPYIN 文節を指定する場合、各変数のマスター・スレッドのコピーまたは *copyin_entity_list* 内に宣言された共通ブロックは、並列領域の始めで複写されます。その並列領域内で実行するチーム内の各スレッドは、 *copyin_entity_list* 内のすべてのエンティティーのプライベート・コピーを受け取ります。*copyin_entity_list* 内に宣言されたすべての変数は、 **THREADPRIVATE** であるか、 **THREADPRIVATE** ディレクティブで使用される共通ブロックのメンバーである必要があります。

構文

```
>>--COPYIN--(--copyin_entity_list--)--------------------------------><
```

copyin_entity

```
>>--+-variable_name-----------+-------------------------------------><
    '-/--common_block_name--/-'
```

variable
: **THREADPRIVATE** 変数、または共通ブロック内の **THREADPRIVATE** 変数です。

common_block_name
: **THREADPRIVATE** 共通ブロック名です。

規則

COPYIN 文節を指定する場合は、以下のことは行えません。

- *copyin_entity_list* の中で同じエンティティー名を複数回指定する。
- 同じディレクティブの別々の **COPYIN** 文節の中で同じエンティティー名を指定する。

- *copyin_entity_list* 内の同じ名前付き共通ブロック内で、共通ブロック名と変数の両方を指定する。
- 同じディレクティブの異なる **COPYIN** 文節の中の同じ名前付き共通ブロック内で、共通ブロック名と変数の両方を指定する。
- **ALLOCATABLE** 属性を持つ変数を指定する。

スレッドのチームのマスター・スレッドが **COPYIN** 文節を含むディレクティブに達すると、スレッドの変数のプライベート・コピーまたは **COPYIN** 文節内に指定される共通ブロックは、マスター・スレッドのコピーと同じ値を持ちます。

COPYIN 文節は、次のディレクティブに適用されます。

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

COPYPRIVATE

COPYPRIVATE 文節を指定する場合は、プライベート変数の値、またはチーム内の 1 つオブジェクトから共用オブジェクトへのポインターの値は、そのチーム内の他のすべてのスレッドに対応する変数にコピーされます。 *copyprivate_entity_list* 内の変数がポインターでない場合は、そのチーム内のすべてのスレッドの対応する変数は、その変数の値を使用して定義されます。変数がポインターの場合は、そのチーム内のすべてのスレッドの対応する変数は、ポインターの関連付け状況を使用して定義されます。整数ポインターおよび想定サイズ配列は、 *copyprivate_entity_list* では使用できません。

構文

```
>>—COPYPRIVATE—(—copyprivate_entity_list—)——————————————————><
```

copyprivate_entity

```
>>——┬—variable——————————————┬——————————————————————————————><
    └—/—common_block_name—/—┘
```

variable 囲んだ並列領域内のプライベート変数です。

common_block_name
: **THREADPRIVATE** 共通ブロック名です。

規則

共通ブロックが *copyprivate_entity_list* に含まれる場合は、 **THREADPRIVATE** ディレクティブ内で指定する必要があります。さらに、 **COPYPRIVATE** 文節は、*object_list* 内のすべての変数が *copyprivate_entity_list* 内に指定されているものとして共通ブロックを扱います。

COPYPRIVATE 文節は、**SINGLE** 構造体の最後にある **END SINGLE** ディレクティブに指定する必要があります。コンパイラーは、スレッドがその構造体の最後にある暗黙の **BARRIER** ディレクティブを通過してしまう前に、 **COPYPRIVATE** 文節を評価します。 *copyprivate_entity_list* で指定された変数は、 **SINGLE** 構造体の **PRIVATE** または **FIRSTPRIVATE** 文節内では使用できません。 **END SINGLE** ディレクティブが並列領域の動的エクステント内で使用された場合、 *copyprivate_entity_list* 内に指定した変数は、その並列領域内でプライベートである必要があります。

COPYPRIVATE 文節は、**NOWAIT** 文節と同じ **END SINGLE** ディレクティブに指定することはできません。

THREADLOCAL 共通ブロックまたはその共通ブロックのメンバーは、**COPYPRIVATE** 文節の一部にすることは許されません。

COPYPRIVATE 文節は、次のディレクティブに適用されます。

- **END SINGLE**

DEFAULT

DEFAULT 文節を指定すると、並列構造体の字句エクステントの中のすべての変数に、*default_scope_attr* の有効範囲属性があるように指定することができます。

DEFAULT(NONE) を指定すると、デフォルトの有効範囲属性は指定されません。したがって、変数が次のものでない限り、並列構造体のデータ有効範囲属性文節の中の並列構造体の字句エクステントの中で使用するそれぞれの変数を明示的にリストする必要があります。

- **THREADPRIVATE**
- **THREADPRIVATE** 共通ブロックのメンバー
- ポインティング先
- 次のもののループ繰り返し変数専用として使用されるループ繰り返し変数
  - 並列領域の字句エクステントの中の順次ループ
  - 並列領域にバインドする並列 DO ループ
- 並列領域にバインドする作業共用構造体の中だけで使用され、それぞれの作業共用構造体のデータ有効範囲属性文節の中で指定される変数

DEFAULT 文節は、並列構造体にあるすべての変数が同じ **PRIVATE**、**SHARED** のどちらかの同じデフォルトの有効範囲属性を共用するか、またはデフォルトの有効範囲属性は使用しないかを指定します。

構文

```
►►──DEFAULT──(──default_scope_attr──)──────────────────────────────────►◄
```

default_scope_attr
: **PRIVATE**、**SHARED**、または **NONE** のうち 1 つです。

規則

ディレクティブに **DEFAULT(NONE)** を指定する場合、すべての名前付き変数と、**FIRSTPRIVATE**、**LASTPRIVATE**、**PRIVATE**、**REDUCTION**、または **SHARED** 文節にあるディレクティブ構造体の字句エクステント内で参照された配列セクション、配列エレメント、構造体コンポーネント、またはサブストリングのすべての左端の名前を指定しなければなりません。

ディレクティブに **DEFAULT(PRIVATE)** を指定する場合、すべての名前付き変数と、ディレクティブ構造体の字句エクステント内で参照された配列セクション、配列エレメント、構造体コンポーネント、またはサブストリングのすべての左端の名前は、共通ブロックおよび使用関連付けされた変数を含み (ただし、**POINTEE** および **THREADLOCAL** 共通ブロックは除く)、**PRIVATE** 文節に明示的にリストされているかのように、スレッドに対して **PRIVATE** 属性を持ちます。

ディレクティブに **DEFAULT(SHARED)** を指定する場合、すべての名前付き変数と、ディレクティブ構造体の字句エクステント内で参照された配列セクション、配列エレメント、構造体コンポーネント、またはサブストリングのすべての左端の名前は、**POINTEE** を除き、 **SHARED** 文節に明示的にリストされているかのように、スレッドに対して **SHARED** 属性を持ちます。

ディレクティブに **DEFAULT** 文節を明示的に示さない場合、デフォルトの動作は、**DEFAULT(SHARED)** になります。

DEFAULT 文節は、次のディレクティブに適用されます。

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

例

次の例は、**DEFAULT(NONE)** の使用法と、並列領域の中の変数のデータ有効範囲属性を指定するための規則のいくつかを示しています。

```
      PROGRAM MAIN
        COMMON /COMBLK/ ABC(10), DEF

          ! THE LOOP ITERATION VARIABLE, I, IS NOT
          ! REQUIRED TO BE IN DATA SCOPE ATTRIBUTE CLAUSE

!$OMP   PARALLEL DEFAULT(NONE) SHARED(ABC)

          ! DEF IS SPECIFIED ON THE WORK-SHARING DO AND IS NOT
          ! REQUIRED TO BE SPECIFIED IN A DATA SCOPE ATTRIBUTE
          ! CLAUSE ON THE PARALLEL REGION.

!$OMP   DO FIRSTPRIVATE(DEF)
        DO I=1,10
          ABC(I) = DEF
        END DO
!$OMP   END PARALLEL
      END
```

IF

IF 文節を指定すると、実行時環境は、ブロックを逐次で実行するのか並列で実行するのかを決定するために、テストを実行します。 *scalar_logical_expression* が真である場合、ブロックは並列で実行され、真でない場合、逐次で実行されます。

構文

```
▶▶─IF─(─scalar_logical_expression─)──────────────────────────────────▶◀
```

規則

PARALLEL SECTIONS 構造体の場合、**PRIVATE** 文節に入っていない変数は、デフォルトで **SHARED** として想定されます。

IF 文節は、任意のディレクティブに 1 度だけ組み込めます。

デフォルトでは、ネストされた並列ループは、**IF** 文節の設定にかかわりなく逐次化されます。このデフォルトは、**-qsmp=nested_par** コンパイラー・オプションを使用して変更できます。

IF 式は、並列構造体のコンテキストの外側で評価されます。 **IF** 式の中の関数参照は、副次作用を与えるものであってはなりません。

IF 文節は、次のディレクティブに適用されます。

- 715 ページの『PARALLEL / END PARALLEL』
- 718 ページの『PARALLEL DO / END PARALLEL DO』
- 722 ページの『PARALLEL SECTIONS / END PARALLEL SECTIONS』
- 726 ページの『PARALLEL WORKSHARE / END PARALLEL WORKSHARE』

FIRSTPRIVATE

FIRSTPRIVATE 文節を指定する場合は、それぞれのスレッドは、*data_scope_entity_list* の中に、独自の初期化されたローカル・コピーと共通ブロックを持っています。

FIRSTPRIVATE 文節は、**PRIVATE** 文節と同じ変数を指定することができ、 **PRIVATE** 文節と同様の方法で関数を指定することができます。例外は、ディレクティブ構造体に入る時の変数の状態です。 **FIRSTPRIVATE** 変数は存在しており、ディレクティブ構造体に入る時にそれぞれのスレッド用に初期化されています。

構文

```
►►──FIRSTPRIVATE──(──data_scope_entity_list──)─────────────────────────►◄
```

規則

FIRSTPRIVATE 文節の変数は、以下のいずれにすることもできません。

- ポインティング先
- 想定サイズ配列
- **THREADLOCAL** 共通ブロック
- **THREADPRIVATE** 共通ブロックまたはそのメンバー
- **THREADPRIVATE** 変数
- 割り振り可能オブジェクト

次の場合、並列構造体の **FIRSTPRIVATE** 文節の中で変数を指定することはできません。

- 変数が、名前リスト・ステートメント、変数形式設定式、またはステートメント関数定義の式の中にある場合。
- 並列構造体内で、形式化された I/O によってステートメント関数、変数形式設定式を参照する場合、または名前リスト I/O によって名前リストを参照する場合。

非同期 I/O 操作を行うエンティティーの 1 つが **FIRSTPRIVATE** 変数、**FIRSTPRIVATE** 変数のサブオブジェクト、または **FIRSTPRIVATE** 変数に関連付けら

れたポインターである場合、一致する暗黙の待機または **WAIT** ステートメントは、スレッドが終わる前に実行されなければなりません。

共通ブロックの個々のメンバーがプライベート化されると、指定された変数のストレージと共通ブロックのストレージとの関連性はなくなります。

FIRSTPRIVATE 変数に関連するストレージである任意の変数は、変列構造体に入るときに未定義になります。

ディレクティブ構造体に、非ブロッキング通信を実行するメッセージ送達インターフェース (MPI) ルーチンへの **FIRSTPRIVATE** 引き数がある場合、MPI 通信は構造体が終わる前に完了する必要があります。

FIRSTPRIVATE 文節は、次のディレクティブに適用されます。

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**
- **SECTIONS**
- **SINGLE**

LASTPRIVATE

LASTPRIVATE 文節を使用する場合は、 *data_scope_entity_list* の中にあるそれぞれの変数と共通ブロックは、 **PRIVATE** であり、 *data_scope_entity_list* の中のそれぞれの変数の最後の値は、ディレクティブの構造体の外側から参照することができます。 **LASTPRIVATE** 文節を **DO** または **PARALLEL DO** に使用する場合は、最後の値はループの最後の繰り返しの後の値になります。 **LASTPRIVATE** 文節を **SECTIONS** または **PARALLEL SECTIONS** に使用する場合は、最後の値は構造体の最後の **SECTION** の後の変数の値になります。 ループの最後の繰り返しまたは構造体の最後のセクションで **LASTPRIVATE** 変数が定義されない場合、変数はループまたは構造体の後でも未定義です。

LASTPRIVATE 文節は、**PRIVATE** 文節と同様の機能を持っているので、同じ基準を満たす変数を指定しなければなりません。 ただし、ディレクティブ構造体から出る時の変数の状況は例外です。コンパイラーは、変数の最後の値を決定して、名前付き変数中に保管されているその値のコピーを取って構造体の後に使用します。 **LASTPRIVATE** 変数は、構造体に入る時に **FIRSTPRIVATE** 変数でない場合には未定義です。

構文

```
►►—LASTPRIVATE—(—data_scope_entity_list—)——————————————————————►◄
```

規則

LASTPRIVATE 文節の変数は、以下のいずれにすることもできません。

- ポインティング先
- 割り振り可能オブジェクト
- 想定サイズ配列
- **THREADLOCAL** 共通ブロック
- **THREADPRIVATE** 共通ブロックまたはそのメンバー
- **THREADPRIVATE** 変数

次の場合、並列構造体の **LASTPRIVATE** 文節の中で変数を指定することはできません。

- 変数が、名前リスト・ステートメント、変数形式設定式、またはステートメント関数定義の式の中にある場合。
- 並列構造体内で、形式化された I/O によってステートメント関数、変数形式設定式を参照する場合、または名前リスト I/O によって名前リストを参照する場合。

非同期 I/O 操作を行うエンティティーの 1 つが **LASTPRIVATE**、**LASTPRIVATE** 変数のサブオブジェクト、または **LASTPRIVATE** 変数に関連付けられたポインターである場合、一致する暗黙の待機または **WAIT** ステートメントは、スレッドが終わる前に実行されなければなりません。

共通ブロックの個々のメンバーがプライベート化されると、指定された変数のストレージと共通ブロックのストレージとの関連性はなくなります。

LASTPRIVATE 変数に関連するストレージである任意の変数は、変列構造体に入るときに未定義になります。

ディレクティブ構造体に、非ブロッキング通信を実行するメッセージ送達インターフェース (MPI) ルーチンへの **LASTPRIVATE** 引き数がある場合、MPI 通信はその構造体が終わる前に完了する必要があります。

変数を作業共用ディレクティブに **LASTPRIVATE** として指定し、そのディレクティブに **NOWAIT** 文節を指定してある場合は、その変数を作業共用構造体の最後と **BARRIER** の間に使用することはできません。

並列構造体に **LASTPRIVATE** として指定する変数は、構造体の最後で定義済みになります。複数のスレッドに同時定義がある場合、または複数のスレッドで **LASTPRIVATE** 変数を使用する場合は、変数が定義済みとなる構造体の最後でスレッドが同期化されるようにする必要があります。たとえば、複数のスレッドが、**LASTPRIVATE** 変数を持つ **PARALLEL** 構造体を検出すると、**LASTPRIVATE** 変数は **END PARALLEL** で定義済みになるので、スレッドが **END PARALLEL** ディレクティブに達するときに、スレッドを同期化する必要があります。したがって、**PARALLEL** 構造体全体は、同期構造体内で完結している必要があります。

LASTPRIVATE 文節は、次のディレクティブに適用されます。

- **DO**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**

例

次の例は、**NOWAIT** 文節の後で **LASTPRIVATE** 変数を使用する正しい方法を示しています。

```
!$OMP PARALLEL
!$OMP   DO LASTPRIVATE(K)

        DO I=1,10
            K=I+1
        END DO

!$OMP   END DO NOWAIT

! PRINT *, K **ERROR** ! The reference to K must occur after a
                       ! barrier.
!$OMP BARRIER
      PRINT *, K       ! This reference to K is legal.
!$OMP END PARALLEL
      END
```

NUM_THREADS

NUM_THREADS 文節を使用すると、並列領域で使用するスレッド数を指定できます。後続の並列領域には反映されません。**NUM_THREADS** 文節は、**omp_set_num_threads** ライブラリー・ルーチンまたは環境変数 **OMP_NUM_THREADS** を使用して指定されたスレッド数よりも優先されます。

構文

```
▶▶─NUM_THREADS─(─scalar_integer_expression─)──────────────────▶◀
```

規則

scalar_integer_expression の値は正でなければなりません。式は、並列領域のコンテキストの外側で評価されます。式の中での関数呼び出し、および式で参照される変数の値を変更する関数呼び出しの結果は、どのようになるか予想できません。

動的スレッドを使用可能にするために環境変数 **OMP_DYNAMIC** を使用する場合は、*scalar_integer_expression* は、並列領域で使用可能なスレッドの最大数を定義します。

ネストされた並列領域の一部として **NUM_THREADS** 文節を含む場合は、**omp_set_nested** ライブラリー・ルーチンを指定するか、または **OMP_NESTED** 環境変数を設定する必要があります。そうしない場合は、その並列領域の実行は直列化されます。

NUM_THREADS 文節は、以下の作業共用構造体に適用されます。

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

ORDERED

作業共用構造体に **ORDERED** 文節を指定すると、並列ループの動的エクステント内に **ORDERED** ディレクティブを指定できます。

構文

```
▶▶─ORDERED────────────────────────────────────────────────────▶◀
```

規則

ORDERED 文節は、次のディレクティブに適用されます。

- 703 ページの『DO / END DO』

- 718 ページの『PARALLEL DO / END PARALLEL DO』

PRIVATE

以下にリストされているディレクティブのいずれか 1 つで **PRIVATE** 文節を指定する場合は、チーム内のそれぞれのスレッドは、 *data_scope_entity_list* の中に、独自の未初期化専用変数と共通ブロックのローカル・コピーを持っています。

変数の値が 1 つのスレッドで計算され、その値が他のスレッドに依存していない場合、または構造体内で使用される前に定義されている場合、またはその値が構造体の終了後には使用されない場合は、変数には **PRIVATE** 属性を指定しなければなりません。 **PRIVATE** 変数のコピーは各スレッドにローカルで存在します。各スレッドは、 **PRIVATE** 変数の、初期化されていない独自のコピーを受け取ります。 **PRIVATE** 変数は、ディレクティブ構造体に入る時と出る時に、未定義の値または関連付け状況を持ちます。 ディレクティブ構造体の字句エクステント内にあるすべてのスレッド変数には、 **PRIVATE** 属性がデフォルトで指定されています。

構文

```
►►──PRIVATE──(──data_scope_entity_list──)─────────────────────────────►◄
```

規則

PRIVATE 文節の変数には、以下のものを指定することはできません。

- ポインティング先
- 想定サイズ配列
- **THREADLOCAL** 共通ブロック
- **THREADPRIVATE** 共通ブロックまたはそのメンバー
- **THREADPRIVATE** 変数

次の場合、並列構造体の **PRIVATE** 文節の中で変数を指定することはできません。

- 変数が、名前リスト・ステートメント、変数形式設定式、またはステートメント関数定義の式の中にある場合。
- 並列構造体内で、形式化された I/O によってステートメント関数、変数形式設定式を参照する場合、または名前リスト I/O によって名前リストを参照する場合。

非同期 I/O 操作を行うエンティティーの 1 つが **PRIVATE** 変数、**PRIVATE** 変数のサブオブジェクト、または **PRIVATE** 変数に関連付けられたポインターである場合、一致する暗黙の待機または **WAIT** ステートメントは、スレッドが終わる前に実行されなければなりません。

共通ブロックの個々のメンバーがプライベート化されると、指定された変数のストレージと共通ブロックのストレージとの関連性はなくなります。

PRIVATE 変数に関連するストレージである任意の変数は、変列構造体に入るときに未定義になります。

ディレクティブ構造体に、非ブロッキング通信を実行するメッセージ送達インターフェース (MPI) ルーチンへの **PRIVATE** 引き数がある場合、MPI 通信はその構造体が終わる前に完了する必要があります。

PRIVATE 文節の *data_scope_entity_list* にある変数名は、割り振り可能オブジェクトであってもかまいません。 このオブジェクトは、ディレクティブ構造体に最初に入る時に割り振られていてはなりません。また、構造体を実行するすべてのスレッドのオブジェクトを割り振りおよび割り振り解除しなければなりません。

ディレクティブ構造体の動的エクステント内にある参照されているサブプログラムの **SAVE** または **STATIC** 属性がないローカル変数には、暗黙の **PRIVATE** 属性があります。

PRIVATE 文節は、次のディレクティブに適用されます。

- **DO**
- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **SECTIONS**
- **SINGLE**
- **PARALLEL WORKSHARE**

例

次の例は、ステートメント関数を定義するために使用される **PRIVATE** 変数の正しい使用法を示しています。注釈行では、無効な使用法が示されています。ステートメント関数の中に *J* があるため、*J* が **PRIVATE** となっている並列構造体内では、ステートメント関数を参照できません。

```
      INTEGER :: ARR(10), J = 17
      ISTFNC() = J

!$OMP PARALLEL DO PRIVATE(J)
      DO I = 1, 10
         J=I
         ARR(I) = J
      !  ARR(I) = ISTFNC() **ERROR**   A reference to ISTFNC would
                                       ! make the PRIVATE(J) clause
```

```
                                             ! invalid.
     END DO
     PRINT *, ARR
     END
```

REDUCTION

REDUCTION 文節は、ディレクティブ構造体内にある文節で宣言された名前付き変数を更新します。 並列構造体内では、**REDUCTION** 変数の中間値は更新自体以外では使用されません。

構文

```
►►──REDUCTION──(──┬─────────────┬──variable_name_list──)──────────────────►◄
                  └─op_fnc──:───┘
```

op_fnc *reduction_op* または *reduction_function* です。これは、この変数も含めて、すべての **REDUCTION** ステートメントに含まれます。ディレクティブ構造体の変数に複数の **REDUCTION** 演算子または関数を指定してはなりません。
OpenMP Fortran API バージョン 2.0 準拠を維持するには、**REDUCTION** 文節に *op_fnc* を指定する必要があります。

REDUCTION ステートメントの形式は、次のいずれかになります。

```
►►──reduction_var_ref──=──expr──reduction_op──reduction_var_ref──────────►◄

►►──reduction_var_ref──=──reduction_var_ref──reduction_op──expr──────────►◄

►►──reduction_var_ref =──reduction_function──(expr,──reduction_var_ref)──►◄

►►──reduction_var_ref =──reduction_function──(reduction_var_ref,──expr)──►◄
```

それぞれの意味は次のとおりです。

reduction_var_ref
: **REDUCTION** 文節に入れる変数または変数のサブオブジェクトです。

reduction_op
: **+**、**-**、*****、**.AND.**、**.OR.**、**.EQV.**、**.NEQV.**、または **.XOR.** のいずれかの組み込み演算子です。

reduction_function
: **MAX**、**MIN**、 **IAND**、**IOR**、または **IEOR** のいずれかの組み込みプロシージャーです。

それぞれの演算子と組み込み機能の正規の初期化値は、次の表に示されています。実際の初期化値は、対応する **REDUCTION** 変数のデータ型と一致します。

組み込み演算子	初期化
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
.XOR.	.FALSE.
組み込みプロシージャー	**初期化**
MAX	表現できる最小の数
MIN	表現できる最大の数
IAND	全ビットがオン
IOR	0
IEOR	0

規則

REDUCTION ステートメントには次の規則が適用されます。

- **REDUCTION** 文節の変数は、**REDUCTION** 文節が入っているディレクティブ構造体内の **REDUCTION** ステートメントにのみ入れることができます。
- **REDUCTION** ステートメントに入れる 2 つの *reduction_var_ref* は、字句的に同一でなければなりません。
- *reduction_var_ref = expr operator reduction_var_ref* の形式の **REDUCTION** ステートメントは使用できません。

共通ブロックの個々のメンバーを **REDUCTION** 文節に指定すると、指定された変数のストレージと共通ブロックとの関連性はなくなります。

作業共用構造体の **REDUCTION** 文節の中で指定する変数は、それを囲んでいる **PARALLEL** 構造体の中で共用される必要があります。

NOWAIT 文節を持つ構造体で **REDUCTION** 文節を使用する場合は、すべてのスレッドが **REDUCTION** 文節を完了したことを確認するためにバリア同期が実行されるまでは、**REDUCTION** 変数は未定義のままです。

REDUCTION 変数は、それが **REDUCTION** 変数として使用された構造体の動的エクステント内で、別の構造体の **FIRSTPRIVATE**、**PRIVATE** または **LASTPRIVATE** 文節の中で使用できません。

REDUCTION 文節に *op_fnc* が指定されている場合、 *variable_name_list* の各変数は組み込みタイプでなければなりません。変数は、ディレクティブ構造体の字句エクステント内の **REDUCTION** ステートメント中でのみ使用できます。 *op_fnc* は、このディレクティブが *trigger_constant* **$OMP** を使用する場合に指定する必要があります。

REDUCTION 文節では縮約演算に入れる名前付き変数を指定します。コンパイラーは、そのような変数のローカル・コピーを保持しますが、構造体から出る時には各コピーを結合します。 **REDUCTION** 変数の中間値は、どのスレッドが計算を最初に終了したかに応じて適当な順序で結合されます。したがって、ある並列実行と別の並列実行とでビットが同一の結果になることは保証できません。これは、複数の並列実行が、同じ数のスレッド、スケジューリング・タイプ、およびチャンク・サイズを使用する場合にも当てはまります。

並列構造体に **REDUCTION** または **LASTPRIVATE** として指定する変数は、構造体の最後で定義済みになります。複数のスレッドに同時定義がある場合、あるいは複数のスレッドで **REDUCTION** または **LASTPRIVATE** 変数を使用する場合は、変数が定義済みとなるときに構造体の最後でスレッドが同期化されるようにする必要があります。たとえば、複数のスレッドが **REDUCTION** 変数を持つ **PARALLEL** 構造体を検出する場合は、**REDUCTION** 変数は **END PARALLEL** で定義済みになるので、スレッドが **END PARALLEL** ディレクティブに達するときに、スレッドを同期化する必要があります。 したがって、**PARALLEL** 構造体全体は、同期構造体内で完結している必要があります。

REDUCTION 文節の変数は、組み込みタイプにすることはできません。 **REDUCTION** 文節にある変数またはエレメントは、次のものにすることはできません。
- ポインティング先
- 想定サイズ配列
- **THREADLOCAL** 共通ブロック
- **THREADPRIVATE** 共通ブロックまたはそのメンバー
- **THREADPRIVATE** 変数
- 割り振り可能オブジェクト
- Fortran 90 ポインター

これらの規則は、OpenMP ディレクティブでの **REDUCTION** の使用法を説明しています。 **REDUCTION** 文節を **INDEPENDENT** ディレクティブで使用している場合は、**INDEPENDENT** ディレクティブを参照してください。

OpenMP での **REDUCTION** 文節は、次のディレクティブに適用されます。
- **DO**

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**
- **SECTIONS**

SCHEDULE

SCHEDULE 文節を使用して、並列化のためのチャンク方式を指定できます。スケジューリング・タイプまたはチャンク・サイズに応じて、異なる方法で作業がスレッドに割り当てられます。

構文

```
►►──SCHEDULE──(──sched_type──┬────┬──)──────────────────────────────────►◄
                             └─,n─┘
```

sched_type
: **AFFINITY**、**DYNAMIC**、**GUIDED**、**RUNTIME**、**STATIC** のいずれか 1 つです。

n
: 正のスカラー整数式でなければなりません。これを **RUNTIME** *sched_type* に対して指定することはしないでください。 *trigger_constant* **$OMP** を使用する場合、スケジューリング・タイプ **AFFINITY** を指定しないでください。

AFFINITY

ループの繰り返しは、最初は *number_of_threads* で指定した数の区画に分割されます (以下の繰り返しを含む)。

```
CEILING(number_of_iterations / number_of_threads)
```

各区画は、最初はスレッドに割り当てられ、*n* が指定されている場合は、その後、さらに *n* 回の繰り返しを含むチャンクに分割されます。 *n* が指定されていない場合は、チャンクは次のループの繰り返しから構成されます。

```
CEILING(number_of_iterations_remaining_in_partition / 2)
```

スレッドが解放されると、スレッドが最初に割り当てられた区画から次のチャンクを取ります。その区画にチャンクが無くなったら、スレッドは、最初に別のスレッドを割り当てた区画から次の利用可能なチャンクを取ります。

アクティブなスレッドは、最初にスリープ・スレッドに割り当てた区分画面の作業を完了させます。

DYNAMIC

n が指定されている場合、ループの繰り返しは、それぞれに *n* 回の繰り返しを含むチャンクに分割されます。 *n* が指定されていない場合は、デフォルトのチャンク・サイズは 1 回の繰り返しになります。

スレッドは、「先入れ先出し」の規則にしたがってこれらのチャンクに割り当てられます。残りの作業のチャンクは、利用可能なスレッドに割り当てられます。この過程はすべての作業が割り当てられるまで続きます。

スリープ状態のスレッドに割り当てられた作業は、アクティブ・スレッドが利用可能になって時点で、そのスレッドが引き継ぎます。

GUIDED

n に値が指定されている場合、ループの繰り返しは、次に続くそれぞれのチャンクのサイズが指数関数的に小さくなるような仕方でチャンクに分割されます。最後のチャンク以外は、*n* は、最小のチャンクのサイズを指定しています。 *n* に値を指定しない場合、デフォルト値は 1 になります。

最初のチャンクのサイズは、次の回数の繰り返しになります。

```
CEILING(number_of_iterations / number_of_threads)
```

後続のチャンクは次の繰り返しで構成されます。

```
CEILING(number_of_iterations_remaining / number_of_threads)
```

それぞれのスレッドがチャンクを終了する時には、それぞれのスレッドは、使用可能な次のチャンクを動的に獲得します。

チーム内の複数のスレッドがばらばらに **DO** 作業共用構造体に達し、それぞれの繰り返しにおける作業量が大体同じであるような状態であれば、ガイド・スケジューリングを使用することができます。 たとえば、**DO** ループの前に、**NOWAIT** 文節のある作業共用 **SECTIONS** または **DO** 構造体が 1 つまたは複数ある場合は、別のスレッドがその最後の繰り返しを実行するのにかかる時間、または k というチャンク・サイズが指定されている場合は、別のスレッドが最後の k 回の繰り返しを実行するのにかかる時間よりも長くバリアで待つスレッドをなくすことができます。 **GUIDED** スケジュールでは、すべてのスケジューリング方式の同期化は最も少なくなります。

n 式は、**DO** 構造体のコンテキストの外側で評価されます。 *n* 式の中の関数参照は、副次作用を与えるものであってはなりません。

SCHEDULE 文節の *n* パラメーターの値は、チーム内のすべてのスレッドについて同じでなければなりません。

RUNTIME

実行時のスケジューリング・タイプを決定します。

実行時に、スケジューリング・タイプは、環境変数 **XLSMPOPTS** を使用して指定できます。その変数を使用してスケジューリング・タイプを指定しない場合、デフォルトのスケジューリング・タイプ **STATIC** が使用されます。

STATIC

n が指定されている場合、ループの繰り返しは、それぞれに *n* 回の繰り返しを含むチャンクに分割されます。各スレッドは、連鎖的にチャンクに割り当てられます。これをブロック巡回スケジューリングといいます。 *n* の値が 1 の場合は、このスケジューリング・タイプを特に巡回スケジューリングといいます。

n が指定されていない場合、チャンクは次の繰り返しを含みます。

```
CEILING(number_of_iterations / number_of_threads)
```

各スレッドは、これらのチャンクのどれか 1 つに割り当てられます。これをブロック巡回スケジューリングといいます。

スリープ状態のスレッドに作業が割り当てられている場合、そのスレッドはスリープ状態を解除され、作業を完了します。

ユーザーがコンパイル時または実行時にスケジューリング・タイプを設定しない場合、**STATIC** がデフォルトのスケジューリング・タイプになります。

規則

特定の **DO** ディレクティブでは、複数の **SCHEDULE** 文節を指定してはなりません。

SCHEDULE 文節は、次のディレクティブに適用されます。

- 703 ページの『DO / END DO』
- 718 ページの『PARALLEL DO / END PARALLEL DO』

SHARED

すべてのセクションは、*data_scope_entity_list* で指定されている同じ変数のコピーと共通ブロックを使用します。

SHARED 文節では、すべてのスレッドから利用できる変数を指定します。変数を **SHARED** として指定すると、ユーザーは、すべてのスレッドにおいて変数の 1 つのコピーを安全に共用できると宣言したことになります。

構文

```
►►──SHARED──(──data_scope_entity_list──)──────────────────────────────►◄
```

data_scope_entity

```
►►─┬─named_variable──────────┬──────────────────────────────────────────►◄
   └─/──common_block_name──/─┘
```

named_variable
ディレクティブ構造体内でアクセスできる名前付き変数

common_block_name
ディレクティブ構造体内でアクセスできる共通ブロック名

規則

SHARED 文節の変数は、以下のいずれにすることもできません。

- ポインティング先
- **THREADLOCAL** 共通ブロック。
- **THREADPRIVATE** 共通ブロックまたはそのメンバー。
- **THREADPRIVATE** 変数。

SHARED 変数、**SHARED** 変数のサブオブジェクト、あるいは **SHARED** 変数と関連付けられたオブジェクト、または **SHARED** 変数のサブオブジェクトを、非組み込みプロシージャーを参照するときの実引き数として入れる場合は、次のようになります。

- 実引き数は、ベクトル添え字を持つ配列セクションです。
- または、実引き数は次のいずれかになります。
  - 配列セクション
  - 想定形状配列、または
  - ポインター配列

  関連付けられた仮引き数は、明示的形状配列または想定サイズ配列になります。

他のスレッドによって仮引き数と関連付けられた共用ストレージへの参照またはこのストレージの定義は、プロシージャー参照と同期化される必要があります。これは、たとえばプロシージャー参照を **BARRIER** の後に置くことによって行うことができます。

SHARED 文節は、次のディレクティブに適用されます。

- **PARALLEL**
- **PARALLEL DO**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

例

次の例では、配列セクション実引き数のあるプロシージャー参照は、関連した仮引き数は明示的形状配列なので、プロシージャー参照をクリティカル・セクションに置くことによって、仮引き数への参照と同期化する必要があります。

```
      INTEGER:: ABC(10)
      I=2; J=5
!$OMP PARALLEL DEFAULT(NONE), SHARED(ABC,I,J)
!$OMP   CRITICAL
          CALL SUB1(ABC(I:J))    ! ACTUAL ARGUMENT IS AN ARRAY
                                 ! SECTION; THE PROCEDURE
                 ! REFERENCE MUST BE IN A CRITICAL SECTION.

!$OMP   END CRITICAL
!$OMP END PARALLEL
      CONTAINS
        SUBROUTINE SUB1(ARR)
          INTEGER:: ARR(1:4)
          DO I=1, 4
            ARR(I) = I
          END DO
        END SUBROUTINE
      END
```

IBM 拡張 の終り

第 14 章 OpenMP 実行環境ルーチンおよびロック・ルーチン

IBM 拡張

OpenMP 仕様は、並列的な実行環境の制御および照会を可能にする多数のルーチンを提供します。

OpenMP インターフェースを介して実行時環境で作成される並列スレッドは、**Fortran Pthreads ライブラリー・モジュール**呼び出しを使用して作成および制御されるスレッドとは別のものと見なされます。以下の説明において「プログラムの直列部分」とは、実行時環境で作成されたいずれか 1 つのスレッドのみによって実行される、プログラムの部分という意味です。たとえば、 **f_pthread_create** を使用して複数のスレッドを作成することができます。 しかし、その後に OpenMP 並列ブロックの外から、または逐次化されネストされた並列領域内から **omp_get_num_threads** を呼び出す場合、この関数は、現在実行中のスレッド数にかかわらず **1** を戻します。

OpenMP 実行環境ルーチンには次のものがあります。

- **omp_get_dynamic**: 775 ページの『omp_get_dynamic』を参照
- **omp_get_max_threads**: 775 ページの『omp_get_max_threads』を参照
- **omp_get_nested**: 776 ページの『omp_get_nested』を参照
- **omp_get_num_procs**: 776 ページの『omp_get_num_procs』を参照
- **omp_get_num_threads**: 777 ページの『omp_get_num_threads』を参照
- **omp_get_thread_num**: 777 ページの『omp_get_thread_num』を参照
- **omp_in_parallel**: 779 ページの『omp_in_parallel』を参照
- **omp_set_dynamic**: 781 ページの『omp_set_dynamic』を参照
- **omp_set_nested**: 783 ページの『omp_set_nested』を参照
- **omp_set_num_threads**: 784 ページの『omp_set_num_threads』を参照

OpenMP 実行時ライブラリーには、ポータブル壁時計タイマーをサポートする 2 つのルーチンが含まれます。 OpenMP タイミング・ルーチンには次のものがあります。

- **omp_get_wtick**: 778 ページの『omp_get_wtick』 を参照
- **omp_get_wtime**: 779 ページの『omp_get_wtime』 を参照

さらに、OpenMP 実行時ライブラリーは、 1 組の単純ロック・ルーチンとネスト可能ロック・ルーチンをサポートします。変数のロックは、これらのルーチンを介してのみ行ってください。単純ロックでは、それらがすでにロック状態にある場合は、ロックされません。単純ロック変数は単純ロックに関連付けられ、単純ロック・ルーチンにのみ渡すことができます。ネスト可能なロックは、同じスレッドによって複数回ロックする

ことができます。ネスト可能ロック変数はネスト可能ロックに関連付けられ、ネスト可能ロック・ルーチンにのみ渡すことができます。

以下に挙げたすべてのルーチンにおいて、ロック変数は、その **KIND** 型付きパラメーターがシンボリック定数 **omp_lock_kind** または **omp_nest_lock_kind** で表される整数です。事前定義されたロック変数は、組み込みデータ型ではない `omp_lib` モジュール内に定義されることに注意してください。

この変数は、コンパイル・モードに従ってサイズ変更されます。この変数は、32 ビット・アプリケーションでは「4」に設定され、64 ビットの非 LDT (Large Data Type) アプリケーションと 64 ビット LDT アプリケーションでは「8」に設定されます。

OpenMP が提供する単純ロック・ルーチンは次のとおりです。
- **omp_destroy_lock**: 『omp_destroy_lock』を参照
- **omp_init_lock**: 780 ページの『omp_init_lock』を参照
- **omp_set_lock**: 782 ページの『omp_set_lock』を参照
- **omp_test_lock**: 784 ページの『omp_test_lock』を参照
- **omp_unset_lock**: 786 ページの『omp_unset_lock』を参照

OpenMP が提供するネスト可能ロック・ルーチンは次のとおりです。
- **omp_destroy_nest_lock**: 775 ページの『omp_destroy_nest_lock』を参照
- **omp_init_nest_lock**: 781 ページの『omp_init_nest_lock』を参照
- **omp_set_nest_lock**: 783 ページの『omp_set_nest_lock』を参照
- **omp_test_nest_lock**: 785 ページの『omp_test_nest_lock』を参照
- **omp_unset_nest_lock**: 786 ページの『omp_unset_nest_lock』を参照

注: ユーザー独自のバージョンの OpenMP ルーチンを定義してインプリメントすることができます。ただし、(**-qnoswapomp** コンパイラー・オプションを指定しない限り) デフォルトでは、他のインプリメントが存在するかどうかにかかわらず、コンパイラーは OpenMP ルーチンの XL Fortran バージョンに置換します。詳細については、「ユーザーズ・ガイド」を参照してください。

omp_destroy_lock

このサブルーチンは、指定されたロック変数とすべてのロックとの関連を解除します。**omp_destroy_lock** 呼び出しによって破棄したロック変数を再びロック変数として使用する前に、 **omp_init_lock** を使って再初期化する必要があります。

注: 初期化されていないロック変数とともに **omp_destroy_lock** を呼び出した場合、その結果は定義できません。

引き数タイプおよび属性

整数タイプの kind **omp_lock_kind**。

例

omp_destroy_lock の使用方法の例は、 780 ページの『omp_init_lock』 を参照してください。

omp_destroy_nest_lock

このサブルーチンは、ロック変数が未定義になる原因となるネスト可能ロック変数を初期化します。変数 *nvar* は、アンロックされ、初期化されたネスト可能ロック変数でなければなりません。

注: 初期化されていない変数を使用して **omp_destroy_nest_lock** を呼び出すと、結果は未定義になります。

引き数タイプおよび属性

整数タイプの kind **omp_nest_lock_kind**。

例

```
USE omp_lib
INTEGER(kind=omp_nest_lock_kind) LOCK
CALL omp_destroy_nest_lock(LOCK)
```

omp_get_dynamic

omp_get_dynamic 関数は、実行時環境による動的スレッド調整が使用可能であれば **.TRUE.** を、そうでなければ **.FALSE.** を戻します。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

デフォルトの論理値

omp_get_max_threads

この関数は、単一の並列領域内で同時に実行できるスレッドの最大数を戻します。戻り値は、**omp_get_num_threads** 関数によって戻される値と同一です。**omp_set_num_threads** を使用してスレッド数を変更した場合、その後に **omp_get_max_threads** を呼び出すと新しい値が戻されます。

この関数にはグローバル・スコープがあります。つまり、これによって戻される最大値は、プログラム内のすべての関数、サブルーチン、およびコンパイル単位に適用されます。この関数は、直列領域と並列領域のどちらで実行しても同じ値を戻します。

omp_set_dynamic に **.TRUE.** と評価される引き数を渡して動的スレッド調整を使用可能にした場合、 **omp_get_max_threads** を使用して、各スレッドに最大サイズのデータ構造を割り振ることができます。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

デフォルトの整数

結果値

単一の並列領域内で同時に実行できるスレッドの最大数。

omp_get_nested

omp_get_nested 関数は、ネストされた並列性が使用可能であれば **.TRUE.** を戻し、ネストされた並列性が使用不可であれば **.FALSE.** を戻します。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

デフォルトの論理値

omp_get_num_procs

omp_get_num_procs 関数は、マシン上のオンライン・プロセッサー数を戻します。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

デフォルトの整数

結果値

マシンにあるプロセッサーの数

omp_get_num_threads

omp_get_num_threads 関数は、その呼び出し元の並列領域を現在実行しているチーム内のスレッド数を戻します。この関数は、それを囲む最も近い **PARALLEL** ディレクティブにバインドします。

チーム内のスレッド数は、 **omp_set_num_threads** サブルーチンおよび **OMP_NUM_THREADS** 環境変数によって制御されます。スレッド数を明示的に設定しない場合、実行時環境は、マシン上のオンライン・プロセッサー数をデフォルトとして使用します。

プログラムの直列部分から、または逐次化されネストされた並列領域から **omp_get_num_threads** を呼び出すと、関数は 1 を戻します。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

デフォルトの整数

結果値

関数の呼び出し元の並列領域を現在実行しているチーム内のスレッド数。

例

```
      USE omp_lib
      INTEGER N1, N2

      N1 = omp_get_num_threads()
      PRINT *, N1
!$OMP PARALLEL PRIVATE(N2)
      N2 = omp_get_num_threads()
      PRINT *, N2
!$OMP END PARALLEL
```

コードの直列セクションでは **omp_get_num_threads** 呼び出しによって 1 が戻されるため、N1 には値 1 が割り当てられます。 N2 には並列領域を実行しているチーム内のスレッド数が割り当てられ、 2 番目の PRINT ステートメントの出力は、 **omp_get_max_threads** の戻り値以下の任意の数になります。

omp_get_thread_num

この関数は、チーム内の現在実行中のスレッドの数を戻します。戻り値は、常に、0 から *NUM_PARTHDS* - 1 になります。*NUM_PARTHDS* は、チーム内で現在実行中のスレッドの数です。チームのマスター・スレッドは 0 の値を戻します。

直列領域内、逐次化されネストされた並列領域、または並列領域の動的エクステントから **omp_get_thread_num** を呼び出すと、関数は値 0 を戻します。

この関数は、最も近い並列領域にバインドします。

結果タイプおよび属性

デフォルトの整数

結果値

0 から *NUM_PARTHDS* - 1 までの、チーム内で現在実行中のスレッドの値。*NUM_PARTHDS* は、チーム内で現在実行中のスレッドの数です。逐次化されネストされた並列領域、または並列領域の動的エクステントからの **omp_get_thread_num** 呼び出しは、0 を戻します。

例

```
      USE omp_lib
      INTEGER NP

!$OMP PARALLEL PRIVATE(NP)
      NP = omp_get_thread_num()
      CALL WORK(NP)
!$OMP MASTER
      NP = omp_get_thread_num()
      CALL WORK(NP)
!$OMP END MASTER
!$OMP END PARALLEL
      END

      SUBROUTINE WORK(THD_NUM)
      INTEGER THD_NUM
      PRINT *, THD_NUM
      END
```

omp_get_wtick

omp_get_wtick 関数は、連続したクロック刻み間の秒数と等しい倍精度値を戻します。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

倍精度実数

結果値

オペレーティング・システム・リアルタイム・クロックの連続する刻みの間の秒数。

例

```
USE omp_lib
DOUBLE PRECISION WTICKS
WTICKS = omp_get_wtick()
PRINT *, 'The clock ticks ', 10 / WTICKS, &
' times in 10 seconds.'
```

omp_get_wtime

omp_get_wtime 関数は、オペレーティング・システムのリアルタイム・クロックの初期値からの秒数と等しい倍精度値を戻します。この値は、プログラムの実行中は変更されないことが保証されます。

omp_get_wtime 関数から戻される値は、チーム内のスレッド全体にわたって均一ではありません。

引き数タイプおよび属性

この関数への引き数はありません。

結果タイプおよび属性

倍精度実数

結果値

オペレーティング・システム・リアルタイム・クロックの初期値以来の秒数。

例

```
      USE omp_lib
      DOUBLE PRECISION START, END
      START = omp_get_wtime()
!     Work to be timed
      END = omp_get_wtime()
      PRINT *, 'Stuff took ', END - START, ' seconds.'
```

omp_in_parallel

omp_in_parallel 関数は、並列で実行している領域の動的エクステントから呼び出されると **.TRUE.** を戻し、それ以外の場合は **.FALSE.** を戻します。 **omp_in_parallel** 呼び出し元の領域が逐次化されていて、並列実行している領域の動的エクステント内でネストされている場合は、この関数は **.TRUE.** を戻します。 (ネストされた並列領域は、デフォルトでは逐次化されます。詳しくは、 783 ページの『omp_set_nested』、および「ユーザーズ・ガイド」の『環境変数 **OMP_NESTED**』を参照してください。)

結果タイプおよび属性

デフォルトの論理値

結果値

並行して実行されている領域の動的エクステントから呼び出された場合は **.TRUE.**。それ以外の場合は **.FALSE.**。

例

```
      USE omp_lib
      INTEGER N, M
      N = 4
      M = 3
      PRINT*, omp_in_parallel()
!$OMP PARALLEL DO
      DO I = 1,N
!$OMP   PARALLEL DO
        DO J=1, M
          PRINT *, omp_in_parallel()
        END DO
!$OMP   END PARALLEL DO
      END DO
!$OMP END PARALLEL DO
```

最初の **omp_in_parallel** の呼び出しでは、 **.FALSE.** が戻されます。これは、いずれかの領域の動的エクステントの外からこれが呼び出されるためです。 2 番目の呼び出しでは、ネストされた **PARALLEL DO** ループが逐次化されていても、 **.TRUE.** が戻されます。これは、外側の **PARALLEL DO** ループの動的エクステント内にこの呼び出しがあるためです。

omp_init_lock

omp_init_lock サブルーチンはロックを初期化して、パラメーターとして渡されたロック変数をそのロックと関連付けます。 **omp_init_lock** を呼び出した後、ロック変数の元の状態はアンロックされます。

注: すでに初期化したロック変数とともにこのルーチンを呼び出すと、その結果は定義できません。

引き数タイプおよび属性

整数の kind **omp_lock_kind**。

例

```
      USE omp_lib
      INTEGER(kind=omp_lock_kind) LCK
      INTEGER ID
```

```
      CALL omp_init_lock(LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      CALL omp_set_lock(LCK)
      PRINT *,'MY THREAD ID IS', ID
      CALL omp_unset_lock(LCK)
!$OMP END PARALLEL
      CALL omp_destroy_lock(LCK)
```

上の例では、1 度に 1 つのスレッドごとに、ロック変数 LCK に関連した所有権を獲得し、スレッド ID を出力して、ロックの所有権を解放します。

omp_init_nest_lock

omp_init_nest_lock サブルーチンを使用すると、ネスト可能ロックを初期化して、指定したロック変数と関連付けることができます。ロック変数の初期状態はアンロックで、初期のネスト・カウントはゼロです。*nvar* の値は、初期化されたネスト可能ロック変数でなければなりません。

注: すでに初期化された変数を使用して **omp_init_nest_lock** を呼び出すと、結果は未定義になります。

引き数タイプおよび属性

整数の kind **omp_nest_lock_kind**。

例

```
      USE omp_lib
      INTEGER(kind=omp_nest_lock_kind) LCK
      CALL omp_init_nest_lock(LCK)
```

omp_init_nest_lock の使用方法を示す例については、 783 ページの『omp_set_nest_lock』を参照してください。

omp_set_dynamic

omp_set_dynamic サブルーチンは、実行時環境における、並列領域の実行に使用できるスレッド数の動的調整を使用可能または使用不可にします。

.TRUE. と評価される *scalar_logical_expression* を指定して **omp_set_dynamic** を呼び出すと、その後の並列領域の実行に使われるスレッド数を実行時環境が自動的に調整することにより、システム・リソースの最適利用を達成できます。**omp_set_num_threads** を使って指定したスレッド数は最大数であり、実際の正確なスレッド数ではありません。

.FALSE. と評価される *scalar_logical_expression* を指定してこのサブルーチンを呼び出すと、スレッド数の動的調整は使用不可になります。実行時環境は、その後の並列領域の実行に使われるスレッド数の調整を自動的に行えません。 **omp_set_num_threads** に渡した値は、実際の正確なスレッド数になります。

デフォルトでは、スレッドの動的調整が使用可能です。ユーザーのコードを正しく実行するために特定の数のスレッドが必要であれば、動的スレッドを明示的に使用不可にしてください。

注: それぞれの並列領域ごとに、スレッド数は固定されます。 **omp_get_num_threads** 関数はその数を戻します。

このサブルーチンは **OMP_DYNAMIC** 環境変数よりも優先されます。

結果タイプおよび属性

デフォルトの論理値

omp_set_lock

omp_set_lock サブルーチンは、指定したロックが使用可能になるまで、呼び出し側スレッドによる次の命令の実行を強制的に待機させます。ロックが使用可能になると、呼び出し側スレッドにその所有権が与えられます。

注: まだ初期化されていないロック変数を指定してこのルーチンを呼び出すと、その結果は未定義です。また、ロックを所有しているスレッドが、 **omp_set_lock** 呼び出しを発行してこれを再びロックしようとすると、デッドロックが発生します。

引き数タイプおよび属性

omp_lock_kind。

例

```
      USE omp_lib
      INTEGER(kind=omp_lock_kind) LCK_X
      CALL omp_init_lock (LCK_X)
!$OMP PARALLEL PRIVATE (I), SHARED (A, X)
!$OMP DO
      DO I = 3, 100
        A(I) = I * 10
        CALL omp_set_lock (LCK_X)
        X = X + A(I)
        CALL omp_unset_lock (LCK_X)
      END DO
!$OMP END DO
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK_X)
```

この例では、共用変数 X の更新時に競合状態を避けるために、ロック変数 LCK_X が使用されています。 X を更新するたびに、更新前にロックを設定し、更新後に設定解除することによって、 1 時点で必ず 1 つのスレッドのみが X を更新しているようにします。

omp_set_nested

omp_set_nested サブルーチンは、ネストされた並列性を使用可能および使用不可にします。

.FALSE. と評価される *scalar_logical_expression* を指定してこのサブルーチンを呼び出すと、ネストされた並列性が使用不可になります。ネストされた並列領域は逐次化され、現在のスレッドによって実行されます。これはデフォルト設定です。

.TRUE. と評価される *scalar_logical_expression* を指定してこのサブルーチンを呼び出すと、ネストされた並列性が使用可能になります。ネストされた並列領域では、追加のスレッドを配置することができます。追加のスレッドを配置するかどうかは、それぞれの実行時環境が決定します。したがって、並列領域の実行に使われるスレッドの数は、ネストされた領域ごとに異なります。

このサブルーチンは、**OMP_NESTED** 環境変数よりも優先されます。

結果タイプおよび属性

デフォルトの論理値

omp_set_nest_lock

omp_set_nest_lock サブルーチンを使用すると、ネスト可能ロックを設定できます。サブルーチンを実行するスレッドは、ロックが使用可能になるまで待ち、使用可能になるとロックを設定してネスト・カウントを増やします。ネスト可能ロックは、サブルーチンを実行するスレッドによって所有される場合に使用可能であり、そうでない場合はアンロックされます。

引き数タイプおよび属性

整数の kind **omp_nest_lock_kind**。

例

```
USE omp_lib
INTEGER P
INTEGER A
INTEGER B
INTEGER ( kind=omp_nest_lock_kind ) LCK

CALL omp_init_nest_lock ( LCK )
```

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + A
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
CALL omp_unset_nest_lock ( LCK )
!$OMP SECTION
CALL omp_set_nest_lock ( LCK )
P = P + B
CALL omp_unset_nest_lock ( LCK )
!$OMP END PARALLEL SECTIONS

CALL omp_destroy_nest_lock ( LCK )
END
```

omp_set_num_threads

omp_set_num_threads サブルーチンは、次の並列領域で使用するスレッドの数を実行時環境に指示します。このサブルーチンに渡される *scalar_integer_expression* は評価されて、その値がスレッド数として使用されます。スレッド数の動的調整 (781 ページの『omp_set_dynamic』を参照) を使用可能にした場合、**omp_set_num_threads** は、次の並列領域で使用するスレッドの最大数を設定します。その後、実際に使用するスレッドの正確な数を実行時環境が決定します。しかし、スレッド数の動的調整を使用不可のにすると、 **omp_set_num_threads** は、次の並列領域で実際に使用するスレッドの正確な数を設定します。

このサブルーチンは、 **OMP_NUM_THREADS** 環境変数よりも優先されます。

注: 並列で実行している領域の動的エクステントからこのサブルーチンを呼び出すと、サブルーチンの動作は未定義になります。

引き数タイプおよび属性

整数

omp_test_lock

omp_test_lock 関数は、指定されたロック変数に関連したロックの設定を試みます。ロックの設定に成功すると **.TRUE.** を戻し、そうでない場合は **.FALSE.** を戻します。いずれの場合も、呼び出しスレッドはプログラムにある後続の命令の実行を継続します。

注: まだ初期化されていないロック変数を指定して **omp_test_lock** を呼び出すと、その結果は未定義になります。

結果タイプおよび属性

デフォルトの論理値

引き数タイプおよび属性

整数の kind **omp_lock_kind**。

結果値

関数がロックを設定できた場合は **.TRUE.**。それ以外の場合は **.FALSE.**。

例

```
      USE omp_lib
      INTEGER LCK
      INTEGER ID
      CALL omp_init_lock (LCK)
!$OMP PARALLEL SHARED(LCK), PRIVATE(ID)
      ID = omp_get_thread_num()
      DO WHILE (.NOT. omp_test_lock(LCK))
        CALL WORK_A (ID)
      END DO
      CALL WORK_B (ID)
      CALL omp_unset_lock (LCK)
!$OMP END PARALLEL
      CALL omp_destroy_lock (LCK)
```

この例では、ロック変数 LCK を設定できるまで、スレッドが WORK_A の実行を繰り返します。ロック変数の設定に成功すると、WORK_B を実行します。

omp_test_nest_lock

omp_test_nest_lock サブルーチンを使用すると、 **omp_set_nest_lock** と同じ方法でロックの設定を試みることができますが、実行スレッドは、ロックが使用可能であることが確認されるまで待機しません。ロックの設定に成功すると、関数はネスト・カウントを増分します。ロックを使用できない場合は、関数はゼロの値を戻します。結果の値は、常にデフォルトの整数です。

引き数タイプおよび属性

整数の kind **omp_nest_lock_kind**。

結果値

関数がロックを設定できた場合は **.TRUE.**。それ以外の場合は **.FALSE.**。

omp_unset_lock

このサブルーチンは、実行中のスレッドに、指定したロックの所有権を解放させせます。この後、そのロックは、必要に応じて、別のスレッドが設定することができるようになります。

注: 以下のいずれかの場合、 **omp_unset_lock** サブルーチンの動作は未定義です。

- 呼び出し側スレッドが、指定されたロックを所有していない、または
- 初期化されていないロック変数を指定してルーチンが呼び出された

結果タイプおよび属性

整数の kind **omp_lock_kind**。

例

omp_unset_lock の使用方法を示す例については、 782 ページの『omp_set_lock』を参照してください。

omp_unset_nest_lock

omp_unset_nest_lock サブルーチンを使用すると、ネスト可能ロックの所有権を解放できます。サブルーチンは、ネスト・カウントを減少させ、関連したスレッドのネスト可能ロックの所有権を解放します。

結果タイプおよび属性

整数の kind **omp_nest_lock_kind**。

例

omp_unset_nest_lock の使用方法を示す例については、 783 ページの『omp_set_nest_lock』を参照してください。

IBM 拡張 の終り

第 15 章 Pthreads ライブラリー・モジュール

IBM 拡張

Pthreads ライブラリー・モジュール (**f_pthread**) とは、AIX pthreads ライブラリーとのインターフェースを容易にするための、データ型とルーチンを定義する Fortran 90 モジュールのことです。 AIX pthreads ライブラリーは、ユーザーのコードを並列化し、スレッド・セーフにするために使用します。**f_pthread** ライブラリー・モジュールの命名規則においては、対応する AIX pthreads ライブラリーのルーチン名またはタイプ定義名の前にサフィックス **f_** を使用します。

AIX バージョン 4.3 以上は、デフォルトの POSIX 1003.1-1996 標準と Draft 7 POSIX pthreads API の両方をサポートします。どちらの呼び出しコマンドを使用するかによって、 POSIX 1003.1-1996 標準または Draft 7 インターフェース・ライブラリーのどちらかで、プログラムのコンパイルとリンクを行うことができます。これを行う方法について詳しくは、「ユーザーズ・ガイド」の『*POSIX pthreads API サポートのレベル*』、『*ld コマンドを使用した 32 ビット SMP オブジェクト・ファイルのリンク*』、および『*ld コマンドを使用した 64 ビット SMP オブジェクト・ファイルのリンク*』を参照してください。

Fortran 90 モジュール **f_pthread** と、AIX pthreads ライブラリーに含まれているライブラリー・ルーチンとの間には、通常では 1 対 1 の対応関係が存在します。しかし、pthreads ルーチンの中には、AIX ではサポートされていないため、対応するプロシージャーがこのモジュール内にないものもあります。そのようなルーチンの一例としては、thread stack address オプションがあります。さらに、**f_pthread** ライブラリー・モジュールの中には、非 pthreads インターフェース・ルーチンもあります。**f_maketime** ルーチンがその一例で、これは **f_timespec** 派生型変数に絶対時間を戻すために組み込まれています。

ルーチンの大半は、整数値を戻します。戻り値 **0** は常に、ルーチンの呼び出しでエラーが発生しなかったことを示します。値がゼロ以外であれば、エラーがあったことを示します。それぞれのエラー・コードには、対応する Fortran のシステム・エラー・コードの定義があります。これらのエラー・コードは Fortran の整数定数として入手することが可能です。 Fortran でのこれらのエラー・コードの命名方法は、対応する AIX エラー・コード名と一貫しています。たとえば、**EINVAL** は、AIX におけるエラー・コード **EINVAL** の Fortran 定数名です。これらのエラー・コードの完全なリストについては、AIX のファイル **/usr/include/sys/errno.h** を参照してください。

Fortran Pthreads ライブラリー呼び出しに対応するシステム呼び出しの詳細については、AIX オペレーティング・システムの資料を参照してください。

Pthreads のデータ構造、関数、およびサブルーチン

Pthreads のデータ構造

- 801 ページのf_pthread_attr_t
- 805 ページのf_pthread_cond_t
- 809 ページのf_pthread_condattr_t
- 815 ページのf_pthread_key_t
- 818 ページのf_pthread_mutex_t
- 825 ページのf_pthread_mutexattr_t
- 826 ページのf_pthread_once_t
- 836 ページのf_pthread_t
- 837 ページのf_sched_param
- 837 ページのf_timespec

スレッド属性オブジェクトに操作を実行する関数

- 790 ページのf_pthread_attr_destroy
- 791 ページのf_pthread_attr_getdetachstate
- 792 ページのf_pthread_attr_getinheritsched
- 792 ページのf_pthread_attr_getschedparam
- 793 ページのf_pthread_attr_getschedpolicy
- 793 ページのf_pthread_attr_getscope
- 794 ページのf_pthread_attr_getstackaddr
- 795 ページのf_pthread_attr_getstacksize
- 795 ページのf_pthread_attr_init
- 796 ページのf_pthread_attr_setdetachstate
- 797 ページのf_pthread_attr_setinheritsched
- 798 ページのf_pthread_attr_setschedparam
- 798 ページのf_pthread_attr_setschedpolicy
- 799 ページのf_pthread_attr_setscope
- 800 ページのf_pthread_attr_setstackaddr
- 800 ページのf_pthread_attr_setstacksize

スレッドに操作を実行する関数およびサブルーチン

- 801 ページのf_pthread_cancel
- 801 ページのf_pthread_cleanup_pop
- 802 ページのf_pthread_cleanup_push
- 809 ページのf_pthread_create
- 811 ページのf_pthread_equal
- 811 ページのf_pthread_exit
- 812 ページのf_pthread_getschedparam
- 813 ページのf_pthread_join
- 815 ページのf_pthread_kill

- 833 ページのf_pthread_self
- 835 ページのf_pthread_setschedparam

mutex 属性オブジェクトに操作を実行する関数

- 819 ページのf_pthread_mutexattr_destroy
- 820 ページのf_pthread_mutexattr_getprioceiling
- 820 ページのf_pthread_mutexattr_getprotocol
- 820 ページのf_pthread_mutexattr_getpshared
- 822 ページのf_pthread_mutexattr_init
- 822 ページのf_pthread_mutexattr_setprioceiling
- 823 ページのf_pthread_mutexattr_setprotocol
- 823 ページのf_pthread_mutexattr_setpshared

mutex オブジェクトに操作を実行する関数

- 816 ページのf_pthread_mutex_destroy
- 816 ページのf_pthread_mutex_getprioceiling
- 816 ページのf_pthread_mutex_init
- 817 ページのf_pthread_mutex_lock
- 818 ページのf_pthread_mutex_setprioceiling
- 818 ページのf_pthread_mutex_trylock
- 819 ページのf_pthread_mutex_unlock

条件変数の属性オブジェクトに操作を実行する関数

- 807 ページのf_pthread_condattr_destroy
- 807 ページのf_pthread_condattr_getpshared
- 808 ページのf_pthread_condattr_init
- 808 ページのf_pthread_condattr_setpshared

条件変数オブジェクトに操作を実行する関数

- 790 ページのf_maketime
- 803 ページのf_pthread_cond_broadcast
- 804 ページのf_pthread_cond_destroy
- 804 ページのf_pthread_cond_init
- 805 ページのf_pthread_cond_signal
- 805 ページのf_pthread_cond_timedwait
- 806 ページのf_pthread_cond_wait

スレッド固有データに操作を実行する関数

- 813 ページのf_pthread_getspecific
- 814 ページのf_pthread_key_create
- 814 ページのf_pthread_key_delete
- 836 ページのf_pthread_setspecific

制御スレッド取り消し機能に操作を実行する関数およびサブルーチン

- 833 ページのf_pthread_setcancelstate
- 834 ページのf_pthread_setcanceltype
- 837 ページのf_pthread_testcancel

1 回限りの初期化の操作を実行する関数

- 825 ページのf_pthread_once

f_maketime

この関数は遅延を秒単位で指定した整数値を受け入れ、絶対時間 (呼び出し時点からの **delay** 秒) を持つ **f_timespec** タイプのオブジェクトを戻します。

結果値

絶対時間 (呼び出し時点からの **delay** 秒) を戻します。

例

```
type(f_timespec) function f_maketime(delay)
    integer(4), intent(in):: delay
end function
```

f_pthread_attr_destroy

この関数は、以前に初期化したスレッド属性オブジェクトが不要になった場合に、そのオブジェクトを破棄する場合に、呼び出す必要があります。その属性オブジェクトで作成されたスレッドは、この処置によって影響を受けることはありません。初期化の時点で割り振られたメモリーは、システムによって最収集されます。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **attr** が無効です。

例

```
integer function f_pthread_attr_destroy(attr)
    type(f_pthread_attr_t), intent(inout):: attr
end function
```

f_pthread_attr_getdetachstate

この関数は、スレッド属性オブジェクト **attr** 内の切り離し状態属性を照会するために使用できます。現行の設定値は、引き数 **detach** によって戻されます。

引き数 **detach** には以下のいずれかの値が入ります。

PTHREAD_CREATE_DETACHED:
この属性設定値のスレッド属性オブジェクトを使って新規スレッドを作成すると、新しく作成されるスレッドは切り離された状態になります。これは、システム・デフォルトです。

PTHREAD_CREATE_UNDETACHED:
この属性設定値のスレッド属性オブジェクトを使って新規スレッドを作成すると、新しく作成されるスレッドは切り離されていない状態になります。

これらのスレッド状態の詳細については、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **attr** が無効です。

例

```
integer function f_pthread_attr_getdetachstate(attr, detach)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: detach
end function
```

f_pthread_attr_getguardsize

この関数は、スレッド属性オブジェクト *attr* 内の *guardsize* 属性を獲得するために使用されます。この属性の現行の設定値は、引き数 *guardsize* によって戻されます。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 attr が無効です。

例

```
integer(4) function f_pthread_attr_getguardsize(attr, guardsize)
   type(f_pthread_attr_t), intent(in):: attr
   integer(kind=REGISTER_SIZE), intent(out):: guardsize
end function f_pthread_attr_getguardsize
```

f_pthread_attr_getinheritsched

この関数は、スレッド属性オブジェクト **attr** 内のスケジューリング特性の継承属性の設定値を照会するのに使用できます。現行の設定値は、引き数 **inherit** によって戻されます。

引き数 **inherit** には以下のいずれかの値が入ります。

PTHREAD_INHERIT_SCHED: 新しく作成されるスレッドは親スレッドのスケジューリング特性を継承し、このスレッドを作成するのに使われたスレッド属性オブジェクトのスケジューリング特性は無視されることを示します。

PTHREAD_EXPLICIT_SCHED: スレッド属性オブジェクトでスケジューリング特性を指定して新しいスレッドを作成すると、作成されたそのスレッドにはこのスケジューリング特性が割り当てられます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL 引き数 **attr** が無効です。

ENOSYS POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

例

```
integer function f_pthread_attr_getinheritsched(attr, inherit)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: inherit
end function
```

f_pthread_attr_getschedparam

この関数は、スレッド属性オブジェクト **attr** 内のスケジューリング特性の設定値を照会するために使用できます。現在の設定値は、引き数 **param** に戻されます。スケジューリング特性の設定値の詳細については、AIX システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL　引き数 **attr** が無効です。

ENOSYS　POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

例

```
integer function f_pthread_attr_getschedparam(attr, param)
    type(f_pthread_attr_t), intent(in):: attr
    type(f_sched_param), intent(out):: param
end function
```

f_pthread_attr_getschedpolicy

この関数は、属性オブジェクト **attr** 内のスケジューリング・ポリシー属性の設定値を照会するために使用できます。スケジューリング・ポリシーの現行の設定値は、引き数 **policy** に戻されます。 AIX における有効なスケジューリング・ポリシーについては、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL
　引き数 **attr** が無効です。

ENOSYS
　POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

例

```
integer function f_pthread_attr_getschedpolicy(attr, policy)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: policy
end function
```

f_pthread_attr_getscope

この関数は、スレッド属性オブジェクト **attr** 内のスケジューリング有効範囲属性の現行設定値を照会するのに使用できます。現行の設定値は、引き数 **scope** によって戻されます。

引き数 **scope** には、以下のいずれかの値が入ります。

PTHREAD_SCOPE_SYSTEM: スレッドは、システム全体にわたるスコープのシステム・リソースと競合します。

PTHREAD_SCOPE_PROCESS: スレッドは、所有側プロセス内のシステム・リソースとローカルに競合します。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL 引き数 **attr** が無効です。

ENOSYS POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

例

```
integer function f_pthread_attr_getscope(attr, scope)
    type(f_pthread_attr_t), intent(in):: attr
    integer(4), intent(out):: scope
end function
```

f_pthread_attr_getstackaddr

この関数は、スレッド属性オブジェクト *attr* 内の *stackaddr* 属性を獲得するために使用されます。この属性の現行の設定値は、引き数 *stackaddr* によって戻されます。引き数 *stackaddr* のタイプは、**整数ポインター**です。 *stackaddr* 属性は、この属性オブジェクトを使用して作成したスレッドのスタック・アドレスを指定します。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

EINVAL

引き数 attr が無効です。

例

```
integer(4) function f_pthread_attr_getstackaddr(attr, stackaddr)
   type(f_pthread_attr_t), intent(in):: attr
   integer(4) int_template
   pointer (stackaddr, int_template)
   intent(out) stackaddr
end function f_pthread_attr_getstackaddr
```

f_pthread_attr_getstacksize

この関数は、属性オブジェクト **attr** 内の現行スタック・サイズ属性の設定値を照会するために使用できます。この関数が正常に実行されると、スタック・サイズがバイト単位で引き数 **ssize** に戻されます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL
引き数 **attr** が無効です。

ENOSYS
POSIX スタック・サイズ・オプションは AIXAIX ではインプリメントされていません。

例

```
integer function f_pthread_attr_getstacksize(attr, ssize)
    type(f_pthread_attr_t), intent(in):: attr
    integer(kind=register_size), intent(out):: ssize
end function
```

f_pthread_attr_init

この関数を呼び出して pthread 属性オブジェクト **attr** を作成および初期化してからでないと、そのオブジェクトは使用できません。 **attr** にはシステム・デフォルトのスレッド属性値が入ります。このオブジェクトを初期化した後は、属性アクセス・プロシージャーを使用して特定の pthread 属性の変更または設定のいずれか (または両方) を実行することができます。この属性オブジェクトを初期化すれば、目的の属性でスレッドを作成するために使用できます。デフォルトの属性の詳細については、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。この関数の実行中にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **attr** が無効です。

ENOMEM この属性オブジェクトを作成するにはメモリーが不十分です。

例

```
integer function f_pthread_attr_init(attr)
    type(f_pthread_attr_t), intent(out):: attr
end function
```

f_pthread_attr_setdetachstate

この関数は、スレッド属性オブジェクト **attr** 内の切り離し状態属性を照会するために使用できます。

引き数 **detach** には以下のいずれかの値が入っていなければなりません。

PTHREAD_CREATE_DETACHED:
この属性設定値のスレッド属性オブジェクトを使って新規スレッドを作成すると、新しく作成されるスレッドは切り離された状態になります。これは、システム・デフォルトの設定値です。

PTHREAD_CREATE_UNDETACHED:
この属性設定値のスレッド属性オブジェクトを使って新規スレッドを作成すると、新しく作成されるスレッドは切り離されていない状態になります。

これらのスレッド状態の詳細については、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **attr** または **detach** が無効です。

例

```
integer function f_pthread_attr_setdetachstate(attr, detach)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: detach
end function
```

f_pthread_attr_setguardsize

この関数は、スレッド属性オブジェクト *attr* にある *guardsize* 属性を設定するために使用します。この属性の新しい値は、引き数 *guardsize* から得ることができます。*guardsize* がゼロの場合、*attr* を使用して作成したスレッドに保護域は提供されません。*guardsize* がゼロより大きい場合、*attr* を使用して作成したそれぞれのスレッドに最小のサイズの *guardsize* バイトの保護域が提供されます。 *guardsize* の詳細については、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のエラー・コードを戻します。

EINVAL
引き数 *attr* または *guardsize* が無効です。

例

```
integer(4) function f_pthread_attr_setguardsize(attr, guardsize)
  type(f_pthread_attr_t), intent(inout):: attr
  integer(kind=REGISTER_SIZE), intent(in):: guardsize
end function f_pthread_attr_setguardsize
```

f_pthread_attr_setinheritsched

この関数は、スレッド属性オブジェクト **attr** 内のスレッド・スケジューリング特性の継承属性を設定するために使用できます。

引き数 **inherit** には以下のいずれかの値が入っていなければなりません。

PTHREAD_INHERIT_SCHED: 新しく作成されるスレッドは親スレッドのスケジューリング特性を継承し、そのスレッドを作成するのに使われたスレッド属性オブジェクトのスケジューリング特性は無視されることを示します。

PTHREAD_EXPLICIT_SCHED: スレッド属性オブジェクトでスケジューリング特性を指定して新しいスレッドを作成すると、作成されたそのスレッドにはこのスケジューリング特性が割り当てられます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL
引き数 **attr** が無効です。

ENOSYS
POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

ENOTSUP
引き数 **inherit** の値はサポートされていません。

例

```
integer function f_pthread_attr_setinheritsched(attr, inherit)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: inherit
end function
```

f_pthread_attr_setschedparam

この関数は、スレッド属性オブジェクト **attr** 内のスケジューリング特性属性を設定するために使用できます。この新たな属性オブジェクトで作成されたスレッドは、それらが作成元のスレッドから継承されたものでなければ、引き数 **param** のスケジューリング特性を想定します。引き数 **param** の sched_priority フィールドは、そのスレッドのスケジューリング優先順位を示します。この優先順位フィールドで想定される値の範囲は 1 ～ 127 でなければなりません。この場合 *127* はスケジューリング優先順位が最も高く、*1* は最も低くなります。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL
引き数 **attr** が無効です。

ENOSYS
POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

ENOTSUP
引き数 **param** の値はサポートされていません。

例

```
integer function f_pthread_attr_setschedparam(attr, param)
    type(f_pthread_attr_t), intent(inout):: attr
    type(f_sched_param), intent(in):: param
end function
```

f_pthread_attr_setschedpolicy

この関数で属性オブジェクトを設定すると、この新たな属性オブジェクトで作成されたスレッドは、それらが作成元のスレッドから継承されたものでなければ、このスケジューリングの設定ポリシーを想定します。

引き数 **policy** には以下のいずれかの定数が入っていなければなりません。

SCHED_FIFO: 先入れ先出し (FIFO) によるスレッドのスケジューリング・ポリシーであることを示します。

SCHED_RR: 周期的スケジューリング・ポリシーであることを示します。

SCHED_OTHER:
デフォルトのスケジューリング・ポリシーです。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL
引き数 **attr** が無効です。

ENOSYS
POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

ENOTSUP
引き数 **policy** の値はサポートされていません。

例

```
integer function f_pthread_attr_setschedpolicy(attr, policy)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: policy
end function
```

f_pthread_attr_setscope

この関数は、スレッド属性オブジェクト **attr** 内の競合有効範囲属性を設定するために使用できます。

引き数 **scope** には以下のいずれかの値が入っていなければなりません。

PTHREAD_SCOPE_SYSTEM: スレッドは、システム全体にわたるスコープのシステム・リソースと競合します。

PTHREAD_SCOPE_PROCESS: スレッドは、所有側プロセス内のシステム・リソースとローカルに競合します。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL
引き数 **attr** が無効です。

ENOTSUP
ENOTSUP は、指定された **scope** が PTHREAD_SCOPE_PROCESS の場合にのみ戻されます。

例

```
integer function f_pthread_attr_setscope(attr, scope)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(4), intent(in):: scope
end function
```

f_pthread_attr_setstackaddr

この関数は、スレッド属性オブジェクト *attr* にある *stackaddr* 属性を設定するために使用します。この属性の新しい値は、引き数 *stackaddr* から得ることができます。引き数 *stackaddr* のタイプは、整数ポインターです。 *stackaddr* 属性は、この属性オブジェクトを使用して作成したスレッドのスタック・アドレスを指定します。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 *attr* が無効です。

例

```
integer(4) function f_pthread_attr_setstackaddr(attr, stackaddr)
  type(f_pthread_attr_t), intent(inout):: attr
  integer(4) int_template
  pointer (stackaddr, int_template)
  intent(in) stackaddr
end function f_pthread_attr_setstackaddr
```

f_pthread_attr_setstacksize

この関数は、pthread 属性オブジェクト **attr** 内のスタック・サイズ属性を設定するために使用できます。引き数 **ssize** は、必要とされるスタック・サイズをバイト単位で示す整数です。この属性オブジェクトを使ってスレッドを作成すると、システムは **ssize** バイトの最小スタック・サイズを割り振ります。

戻りコード

この関数の実行中にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **attr** または **ssize** が無効です。

例

```
integer function f_pthread_attr_setstacksize(attr, ssize)
    type(f_pthread_attr_t), intent(inout):: attr
    integer(kind=register_size), intent(in):: ssize
end function
```

f_pthread_attr_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければなりません。

このデータ型は POSIX の **pthread_attr_t** に相当します。これは、スレッド属性オブジェクトのタイプです。

f_pthread_cancel

この関数は、ターゲットのスレッドを取り消すために使用できます。この取り消し要求がどのように処理されるかは、ターゲットのスレッドが持つ取り消し機能の状態によって異なります。ターゲットのスレッドは、引き数 **thread** によって識別されます。ターゲットのスレッドが遅延取り消し状態になっている場合は、ターゲットのスレッドが次の取り消し点に達するまで、この取り消し要求は保留されたままになります。ターゲットのスレッドがその取り消し機能を無効にしている場合は、それが再び有効になるまでこの要求は保留されたままになります。ターゲットのスレッドが非同期取り消し状態にある場合は、この要求は即時実行されます。スレッドの取り消しに関する詳細と、セキュリティーに関する考慮事項については、AIX オペレーティング・システム資料を参照してください。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL
引き数 **thread** が無効です。

例

```
integer function f_pthread_cancel(thread)
    type(f_pthread_t), intent(inout):: thread
end function
```

f_pthread_cleanup_pop

このサブルーチンは、スレッド・セーフティー用のクリーンアップ・スタックを使用するにあたって、**f_pthread_cleanup_push** と組み合わせて使わなければなりません。提供されている引き数 **exec** にゼロ以外の値が入っている場合、最後にプッシュされたクリーンアップ関数がクリーンアップ・スタックからポップされ、そのクリーンアップ関数に渡された引き数 **arg** (最後の **f_pthread_cleanup_push** からの) を指定して実行されます。

ベクトル添え字を持つ配列セクションを引き数 **arg** に渡すと、結果は予測不能です。

実引き数 **arg** は変数でなければならず、代入ステートメントの左に代入できなければなりません。

exec にゼロ値が入っている場合は、最後にプッシュされたクリーンアップ関数はクリーンアップ・スタックからポップされますが、実行はされません。

戻りコード

このサブルーチンからの戻り値はありません。

例

```
subroutine f_pthread_cleanup_pop(exec)
    integer(4), intent(in):: exec
end subroutine
```

f_pthread_cleanup_push

この関数は、呼び出しスレッドにクリーンアップ・サブルーチンを登録するために使用できます。呼び出しスレッドの予期しない終了に対しては、その呼び出しスレッドが安全に終了できるように、システムが自動的にクリーンアップ・サブルーチンを実行します。引き数 **cleanup** は、ただ 1 つの引き数を予期するサブルーチンでなければなりません。それが実行される場合、引き数 **arg** は実引き数としてそのサブルーチンに渡されます。

引き数 **arg** は汎用引き数であり、任意のタイプやランクにすることができます。実引き数 arg は変数でなければならず、代入ステートメントの左に代入できなければなりません。ベクトル添え字を持つ配列セクションを引き数 **arg** に渡すと、結果は予測不能です。

実引き数 **arg** が配列セクションである場合、サブルーチン cleanup の対応する仮引き数は想定形状配列でなければなりません。そうでない場合、結果は予測できません。

実引き数 **arg** が、配列または配列セクションを指し示すポインター属性を持っている場合、サブルーチン cleanup 内の対応する仮引き数は、ポインター属性を持っているか、または想定形状配列でなければなりません。そうでない場合、結果は予測できません。

通常の実行パスの場合、この関数は **f_pthread_cleanup_pop** への呼び出しと組み合わせて使わなければなりません。

引き数 **flag** は、引き数 **arg** の特性を正確にシステムに伝えるために使用しなければなりません。

引き数 **flag** には、以下の定数のいずれか、あるいは以下の定数を組み合わせたものを値として指定できます。

FLAG_CHARACTER:
入り口サブルーチン **cleanup** がタイプ **CHARACTER** の引き数を、方法や形式に関係なく予期している場合は、このフラグ値を組み込んでそのことを示さなければなりません。ただし、サブルーチンが予期しているのが、タイプ **CHARACTER** の引き数を指し示した Fortran 90 ポインターであれば、**FLAG_DEFAULT** 値を代わりに組み込む必要があります。

FLAG_ASSUMED_SHAPE:
入り口サブルーチン **cleanup** が仮引き数を持ち、それが任意のランクの、想定形状配列である場合は、このフラグ値を組み込んでそのことを示さなければなりません。

FLAG_DEFAULT:
上記以外の場合には、このフラグ値が必要です。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

ENOMEM
システムは、このルーチンをプッシュするためのメモリーを割り振ることができません。

EAGAIN
システムは、このルーチンをプッシュするためのリソースを割り振ることができません。

EINVAL
引き数 **flag** が無効です。

例

```
integer function f_pthread_cleanup_push(cleanup, flag, arg)
    external cleanup
    integer(4), intent(in):: flag
end function
```

f_pthread_cond_broadcast

この関数は、条件変数 **cond** を待機しているすべてのスレッドを非ブロック化します。この条件変数を待機しているスレッドがない場合、関数は正常に実行されますが、**f_pthread_cond_wait** への次の呼び出し元はブロックされ、条件変数 **cond** を待機することになります。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL
引き数 **cond** が無効です。

例

```
integer function f_pthread_cond_broadcast(cond)
    type(f_pthread_cond_t), intent(inout):: cond
end function
```

f_pthread_cond_destroy

この関数は、すでに不要となっている条件変数を破棄するために使用できます。ターゲットの条件変数は、引き数 **cond** により識別されます。初期化の間に割り振られたシステム・リソースは、システムによって再収集されます。スレッドの同期化や条件変数の使用方法について詳しくは、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EBUSY 条件変数 **cond** は、現在他のスレッドが使用中です。

EINVAL 引き数 **cond** が無効です。

例

```
integer function f_pthread_cond_destroy(cond)
    type(f_pthread_cond_t), intent(inout):: cond
end function
```

f_pthread_cond_init

この関数は、条件変数 **cond** を動的に初期化するために使用できます。その属性は、属性オブジェクト **cattr** が提供されている場合はそれに従って設定されます。この属性オブジェクトが提供されていなければ、その属性はシステム・デフォルトに設定されます。条件変数が正常に初期化された後は、それを使ってスレッドを同期化できます。スレッドの同期化や条件変数の使用方法について詳しくは、AIX オペレーティング・システムの資料を参照してください。

条件変数を初期化する別の方法は、Fortran 定数 **PTHREAD_COND_INITIALIZER** を使って静的に初期化する方法です。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EBUSY 条件変数がすでに使用中です。これは初期化されており、破棄されてはいません。

EINVAL 引き数 **cond** または **cattr** が無効です。

例

```
integer function f_pthread_cond_init(cond, cattr)
    type(f_pthread_cond_t), intent(out):: cond
    type(f_pthread_condattr_t), intent(in), optional:: cattr
end function
```

f_pthread_cond_signal

この関数は、条件変数 **cond** を待機している最低 1 つのスレッドを非ブロック化します。この条件変数を待機しているスレッドがない場合、関数は正常に実行されますが、**f_pthread_cond_wait** への次の呼び出し元はブロックされ、条件変数 **cond** を待機することになります。 スレッドの同期化や条件変数の使用方法について詳しくは、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **cond** が無効です。

例

```
integer function f_pthread_cond_signal(cond)
    type(f_pthread_cond_t), intent(inout):: cond
end function
```

f_pthread_cond_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介して操作しなければなりません。さらに、このタイプのオブジェクトは Fortran 定数 **PTHREAD_COND_INITIALIZER** を使ってコンパイル時に初期化できます。

このデータ型は POSIX の **pthread_cond_t** に相当します。これは、条件変数オブジェクトのタイプです。

f_pthread_cond_timedwait

この関数は、特定の条件が発生するのを待機するために使用できます。引き数 **mutex** は、この関数を呼び出す前にロックしていなければなりません。 mutex のロックは自動的に解除され、呼び出しスレッドは条件が発生するのを待機します。引き数 **timeout** には、条件が発生するまでの期限を指定します。条件が発生する前に期限に達すると、関数はエラー・コードを戻します。この関数は、それが有効な状態であれば、呼び出しスレッドを取り消すことができる取り消し点を提供します。

引き数 **timeout** には、Oct. 31 10:00:53, 1998 の形式で絶対日付を指定します。関連情報については、**f_maketime** および **f_timespec** の項を参照してください。絶対日付については、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL　引き数 **cond**、**mutex**、または **timeout** が無効です。

EDEADLK　引き数 **mutex** が、呼び出しスレッドによってロックされていません。

ETIMEDOUT　条件が発生する前に、待機の期限が満了しました。

例

```
integer function f_pthread_cond_timedwait(cond, mutex, timeout)
    type(f_pthread_cond_t), intent(inout):: cond
    type(f_pthread_mutex_t), intent(inout):: mutex
    type(f_timespec), intent(in):: timeout
end function
```

f_pthread_cond_wait

この関数は、特定の条件が発生するのを待機するために使用できます。引き数 **mutex** は、この関数を呼び出す前にロックしていなければなりません。 mutex のロックは自動的に解除され、呼び出しスレッドは条件が発生するのを待機します。該当する条件が発生しないと、この関数は呼び出しスレッドが別の方法で終了するまで待機することになります。この関数は、それが有効な状態であれば、呼び出しスレッドを取り消すことができる取り消し点を提供します。

戻りコード

この関数が正常に実行されると、関数が戻される前に mutex が再びロックされます。この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL　引き数 **cond** または **mutex** が無効です。

EDEADLK　**mutex** が、呼び出しスレッドによってロックされていません。

例

```
integer function f_pthread_cond_wait(cond, mutex)
    type(f_pthread_cond_t), intent(inout):: cond
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_condattr_destroy

この関数を呼び出せば、すでに不要となっている条件変数属性オブジェクトを破棄できます。ターゲットのオブジェクトは、引き数 **cattr** によって識別されます。初期化の時点で割り振られたシステム・リソースは再収集されます。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **cattr** が無効です。

例

```
integer function f_pthread_condattr_destroy(cattr)
  type(f_pthread_condattr_t), intent(inout):: cattr
end function
```

f_pthread_condattr_getpshared

この関数は、引き数 **cattr** によって識別される条件変数属性オブジェクトのプロセス共用属性を照会するために使用できます。この属性の現行の設定値は、引き数 **pshared** に戻されます。 **pshared** には以下のいずれかの値が入ります。

PTHREAD_PROCESS_SHARED この条件変数は、異なるプロセスに属するスレッドがメモリーに割り当てられている場合でも、メモリーにアクセスしているすべてのスレッドによって使用することができます。

PTHREAD_PROCESS_PRIVATE 条件変数が使用されるのは、プロセスを作成したスレッドと同じプロセス内でのみです。

戻りコード

この関数が正常に完了すると、値 0 を戻します。さらに、プロセス共用属性が、引き数 **pshared** を介して戻されます。正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 **cattr** が無効です。

例

```
integer(4) function f_pthread_condattr_getpshared(cattr, pshared)
  type(f_pthread_condattr_t), intent(in):: cattr
  integer(4), intent(out):: pshared
end function f_pthread_condattr_getpshared
```

f_pthread_condattr_init

この関数は、インプリメンテーションで定義されたすべての属性に対してデフォルト値を使用して条件変数属性オブジェクト cattr を初期化するために使用します。すでに初期化されている条件変数属性オブジェクトを初期化しようとすると、未定義な振る舞いが起こります。条件変数属性オブジェクトを使用して 1 つ以上の条件変数を初期化した後、属性オブジェクトに影響 (消滅を含む) する関数は、前に初期化された条件変数には影響を与えません。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

ENOMEM
条件変数属性オブジェクトを初期化するための十分なメモリーがありません。

例

```
integer function f_pthread_condattr_init(cattr)
  type(f_pthread_condattr_t), intent(inout):: cattr
end function
```

f_pthread_condattr_setpshared

この関数は、引き数 **cattr** によって識別される条件変数属性オブジェクトのプロセス共用属性を設定するために使用できます。このプロセス共用属性は、引き数 **pshared** に従って設定されます。**pshared** には以下のいずれかの値が入っていなければなりません。

PTHREAD_PROCESS_SHARED この条件変数は、異なるプロセスに属するスレッドがメモリーに割り当てられている場合でも、メモリーにアクセスしているすべてのスレッドによって使用することができます。

PTHREAD_PROCESS_PRIVATE 条件変数が使用されるのは、プロセスを作成したスレッドと同じプロセス内でのみであることを指定します。これは、属性のデフォルト設定です。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のエラー・コードを戻します。

EINVAL
引き数 **cattr** または **pshared** で指定された値が無効です。

例

```
integer(4) function f_pthread_condattr_setpshared(cattr, pshared)
  type(f_pthread_condattr_t), intent(inout):: cattr
  integer(4), intent(in):: pshared
end function f_pthread_condattr_setpshared
```

f_pthread_condattr_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければなりません。

このデータ型は POSIX の **pthread_condattr_t** に相当します。これは、条件変数属性オブジェクトのタイプです。

f_pthread_create

この関数は、現行プロセスに新しいスレッドを作成するために使用します。新しく作成されるスレッドは、スレッド属性オブジェクト **attr** が提供されていれば、その中で定義されている属性を想定します。そうでない場合は、新しいスレッドはシステム・デフォルトの属性を持ちます。新しいスレッドは、サブルーチン **ent** から実行を開始します。それには 1 つの仮引き数を指定することが必要です。システムは入り口サブルーチン **ent** に引き数 **arg** をその実引き数として渡します。引き数 **flag** は、システムに引き数 **arg** の特性を知らせるために使われます。実行が入り口サブルーチン **ent** から戻ると、その新しいスレッドは自動的に終了します。

サブルーチン **ent** を直接呼び出す場合明示インターフェースが必要となるようにこのサブルーチンを宣言した場合、それが引き数としてこの関数に渡される時点においても明示インターフェースが必要となります。

引き数 **arg** は汎用引き数であり、任意のタイプやランクにすることができます。実引き数 **arg** は変数でなければならず、代入ステートメントの左に代入できなければなりません。ベクトル添え字を持つ配列セクションを引き数 **arg** に渡す場合、結果は予測不能です。

実引き数 **arg** が配列セクションである場合、サブルーチン **ent** の対応する仮引き数は想定形状配列でなければなりません。そうでない場合、結果は予測できません。

実引き数 **arg** が、配列または配列セクションを指し示すポインター属性を持っている場合、サブルーチン **ent** 内の対応する仮引き数は、ポインター属性を持っているか、または想定形状配列でなければなりません。そうでない場合、結果は予測できません。

引き数 **flag** は、引き数 **arg** の特性を正確にシステムに伝えるために使用しなければなりません。**flag** には、以下の定数のいずれか、あるいは以下の定数を組み合わせたものを値として指定できます。

FLAG_CHARACTER:
入り口サブルーチン **ent** がタイプ **CHARACTER** の引き数を、方法や形式に関係なく予期している場合は、このフラグ値を組み込んでそのことを示さなければなりません。ただし、サブルーチンが予期しているのが、タイプ **CHARACTER** の引き数を指し示した Fortran 90 ポインターであれば、**FLAG_DEFAULT** 値を代わりに組み込む必要があります。

FLAG_ASSUMED_SHAPE:
入り口サブルーチン **ent** が仮引き数を持ち、それが任意のランクの、想定形状配列である場合は、このフラグ値を組み込んでそのことを示さなければなりません。

FLAG_DEFAULT:
上記以外の場合には、このフラグ値が必要です。

戻りコード

この関数への呼び出しが正常に行われると、新しく作成されたスレッドの ID が引き数 **thread** によって戻されます。そうでない場合には、以下のいずれかのエラー・コードを戻します。

EAGAIN
新しいスレッドを作成するのに十分なリソースがシステムにありません。

EINVAL
引き数 **thread**、**attr**、または **flag** が無効です。

ENOMEM
新しいスレッドを作成するのに十分なメモリーがシステムにありません。

例

```
integer function f_pthread_create(thread, attr, flag, ent, arg)
    type(f_pthread_t), intent(out):: thread
    type(f_pthread_attr_t), intent(in), optional:: attr
    integer(4), intent(in):: flag
    external ent
end function
```

f_pthread_detach

この関数は、引き数 thread に指定されたスレッドの記憶域が、このスレッドが終了するときに要求されることを pthreads ライブラリー・インストール・システムに示すために使用されます。スレッドが終了していない場合、**f_pthread_detach** はそれが終了しないようにします。同じターゲット・スレッド上で **f_pthread_detach** を複数回呼び出すと、エラーが生じます。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

EINVAL
: 引き数 **thread** が無効です。

例

```
integer(4) function f_pthread_detach(thread)
  type(f_pthread_t), intent(in):: thread
end function f_pthread_detach
```

---

f_pthread_equal

この関数は、2 つのスレッド ID が同一のスレッドを識別するかどうかを比較するのに使用できます。

戻りコード

TRUE 該当する 2 つのスレッド ID が同一のスレッドを識別します。

FALSE 該当する 2 つのスレッド ID が同一のスレッドを識別しません。

例

```
logical function f_pthread_equal(thread1, thread2)
  type(f_pthread_t), intent(in):: thread1, thread2
end function
```

---

f_pthread_exit

このサブルーチンを明示的に呼び出して、入り口サブルーチンから戻る前に呼び出しスレッドを終了させることができます。行われる処置は、呼び出しスレッドの状態によって異なります。切り離し状態以外の状態であれば、呼び出しスレッドは結合されるまで待機します。スレッドが切り離し状態である場合、あるいは、それが他のスレッドによって結合されている場合は、呼び出しスレッドは安全に終了します。最初にスタックがポップおよび実行され、その後、スレッド固有データがデストラクターにより破棄されます。最後に、スレッドのリソースが解放されて、結合しているスレッドに引き数 **ret** が戻されます。このサブルーチンの引き数 **ret** はオプションです。現時点では、引き数 **ret** は整数ポインターに制限されています。これが整数ポインターでないと、その動作は不定です。

戻りコード

このサブルーチンが戻ることはありません。引き数 **ret** を指定しないと、このスレッドの出口状況として NULL が指定されます。

例

```
subroutine f_pthread_exit(ret)
    pointer(ret, byte)
    optional ret
    intent(in) ret
end subroutine
```

f_pthread_getconcurrency

この関数は、前の **f_pthread_setconcurrency** 関数への呼び出しによって設定された並行性レベルの値を戻します。前に **f_pthread_setconcurrency** 関数が呼び出されていない場合、この関数はゼロを戻し、システムが現在のレベルを維持していることを示します。並行性レベルの詳細については、 AIX オペレーティング・システム資料を参照してください。

例

```
integer(4) function f_pthread_getconcurrency()
end function f_pthread_getconcurrency
```

f_pthread_getschedparam

この関数は、ターゲットとなるスレッドのスケジューリング特性の現行設定値を照会するために使用できます。ターゲットのスレッドは、引き数 **thread** によって識別されます。スケジューリング・ポリシーは引き数 **policy** によって戻され、スケジューリング特性は引き数 **param** によって戻されます。**param** 内の sched_priority フィールドは、スケジューリング優先順位を定義します。この優先順位フィールドで想定される値の範囲は 1 〜 127 です。この場合 *127* はスケジューリング優先順位が最も高く、*1* は最も低くなります。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

ENOSYS
POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。

ESRCH
ターゲットのスレッドが存在しません。

例

```
integer function f_pthread_getschedparam(thread, policy, param)
    type(f_pthread_t), intent(in):: thread
    integer(4), intent(out):: policy
    type(f_sched_param), intent(out):: param
end function
```

f_pthread_getspecific

この関数は、**key** に関連したスレッド固有データを検索するために使用できます。この関数がスレッド固有データを戻す場合、引き数 **arg** はオプションではないことに注意してください。プロシージャーの実行後、引き数 **arg** はデータへのポインターを保持するか、検索するデータがない場合は **NULL** を保持します。引き数 **arg** は整数ポインターでなければなりません。そうでない場合の結果は未定義です。

実引き数 **arg** は変数でなければならず、代入ステートメントの左に代入できなければなりません。ベクトル添え字を持つ配列セクションを引き数 **arg** に渡す場合、結果は予測不能です。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **key** が無効です。

例

```
integer function f_pthread_getspecific(key, arg)
    type(f_pthread_key_t), intent(in):: key
    pointer(arg, byte)
    intent(out) arg
end function
```

f_pthread_join

この関数を呼び出せば、引き数 **thread** で指定した特定のスレッドを結合できます。ターゲットのスレッドが切り離し以外の状態にあってすでに終了している場合、この呼び出しはすぐに戻され、引き数 **ret** が指定されていれば、ターゲットのスレッドの状況がその中に戻されます。引き数 **ret** はオプションです。現時点では、**ret** を指定する場合は整数ポインターでなければなりません。

ターゲットのスレッドが切り離し状態にある場合は、それを結合するのはエラーとなります。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EDEADLK この呼び出しはデッドロックの原因となります。あるいは、呼び出しスレッドが自分自身を結合しようとします。

EINVAL 引き数 **thread** が無効です。

ESRCH 引き数 **thread** で指定されているスレッドが存在していないか切り離し状態にあります。

例

```
integer function f_pthread_join(thread, ret)
    type(f_pthread_t), intent(in):: thread
    optional ret
    intent(out) ret
    pointer(ret, byte)
end function
```

f_pthread_key_create

この関数は、スレッド固有データ・キーを獲得するために使用できます。該当するキーは引き数 **key** に戻されます。引き数 **dtr** は、この呼び出し点の後にスレッドが終了する時点で、このキーに関連したスレッド固有データを破壊するために使われるサブルーチンです。該当するデストラクターは、スレッド固有データをその引き数として受け取ります。デストラクターそのものはオプションです。これが指定されていないと、システムはこのキーに関連したスレッド固有データに対して、デストラクターを呼び出しません。スレッド固有データのキーの数が、プロセスごとに制限されていることに注意してください。キーの使用を管理するのはユーザーの責任です。プロセスごとの制限については、Fortran 定数 **PTHREAD_DATAKEYS_MAX** で確認できます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EAGAIN キーの最大数を超えています。

EINVAL 引き数 **key** が無効です。

ENOMEM このキーを作成するにはメモリーが不十分です。

例

```
integer function f_pthread_key_create(key, dtr)
    type(f_pthread_key_t), intent(out):: key
    external dtr
    optional dtr
end function
```

f_pthread_key_delete

この関数は、引き数 **key** によって識別されるスレッド固有データのキーを破棄します。該当するキーに関連したスレッド固有データがないことを確認するのはユーザーの責任です。この関数は、スレッドに代わってデストラクターを呼び出すことはしません。キーを破棄した後は、システムはそれを **f_pthread_key_create** 要求に対して再利用できます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL 引き数 **key** が無効です。

EBUSY このキーに関連したデータがまだ存在します。

例

```
integer function f_pthread_key_delete(key)
    type(f_pthread_key_t), intent(inout):: key
end function
```

f_pthread_key_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければなりません。

このデータ型は POSIX の **pthread_key_t** に相当します。これは、スレッド固有データにアクセスするためのキー・オブジェクトのタイプです。

f_pthread_kill

この関数は、ターゲットのスレッドにシグナルを送信するために使用できます。ターゲットのスレッドは、引き数 **thread** によって識別されます。ターゲットのスレッドに送られるシグナルは、引き数 **sig** で識別されます。 **sig** に入っている値がゼロの場合、システムはエラー検査を実行しますが、シグナルは送信されません。マルチスレッド・システムにおけるシグナルの管理の詳細については、 AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL 引き数 **thread** または **sig** が無効です。

ESRCH ターゲットのスレッドが存在しません。

例

```
integer function f_pthread_kill(thread, sig)
    type(f_pthread_t), intent(inout):: thread
    integer(4), intent(in):: sig
end function
```

f_pthread_mutex_destroy

すでに不要になっている mutex オブジェクトを破棄するには、この関数を呼び出す必要があります。システムはこの方法でメモリー・リソースを再収集します。ターゲットの mutex オブジェクトは、引き数 **mutex** によって識別されます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EBUSY
ターゲットの mutex が他のスレッドによりロックまたは参照されています。

EINVAL
引き数 **mutex** が無効です。

例

```
integer function f_pthread_mutex_destroy(mutex)
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_mutex_getprioceiling

この関数は、引き数 **mutex** によって識別される mutex オブジェクトの優先順位上限属性を動的に照会するために使用できます。現行の上限値は、引き数 **old** によって戻されます。

戻りコード

mutex 優先順位保護プロトコルが AIX にインプリメントされていないため、この関数は現在使用できないことに注意してください。

例

```
integer function f_pthread_mutex_getprioceiling(mutex, old)
  type(f_pthread_mutex_t), intent(in):: mutex
  integer(4), intent(out):: old
end function
```

f_pthread_mutex_init

この関数は、引き数 **mutex** によって識別される mutex オブジェクトを初期化するために使用できます。初期化された mutex は、mutex 属性オブジェクト **mattr** がある場合には、それに設定されている属性を想定します。 **mattr** が提供されていないと、システムはデフォルトの属性を持つように mutex を初期化します。 mutex オブジェクトは初期化後に、クリティカルなデータやコードへのアクセスを同期化するために使用できます。さらに、より複雑なスレッド同期オブジェクトを作成する場合にも使用できます。

mutex オブジェクトを初期化する別の方法は、Fortran 定数 **PTHREAD_MUTEX_INITIALIZER** を介してそれらを静的に初期化する方法です。初期化にあたってこの方法を利用すれば、mutex オブジェクトを使用する前にこの関数を呼び出す必要がありません。

戻りコード

この関数の実行時にエラーが発生すると、以下のいずれかのエラー・コードを戻します。

EAGAIN
この mutex を初期化するのに十分なリソースがシステムにありませんでした。

EBUSY
この mutex はすでに使用中です。これは初期化されており、破棄されてはいませんでした。

EINVAL
引き数 **mutex** または **mattr** が無効です。

ENOMEM
この mutex を初期化するにはメモリーが不十分です。

例

```
integer function f_pthread_mutex_init(mutex, mattr)
    type(f_pthread_mutex_t), intent(out):: mutex
    type(f_pthread_mutexattr_t), intent(in), optional:: mattr
end function
```

f_pthread_mutex_lock

この関数は、mutex オブジェクトの所有権を獲得するために使用できます。 (つまり、この関数はその mutex をロックします。) mutex がすでに他のスレッドによってロックされている場合は、呼び出し元はその mutex がアンロックされるまで待機します。その mutex が呼び出し元自身によってすでにロックされている場合は、再帰的ロックを防止するためにエラーが戻されます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EDEADLK
該当する mutex が、呼び出しスレッドによってすでにロックされています。

EINVAL
引き数 **mutex** が無効です。

例

```
integer function f_pthread_mutex_lock(mutex)
  type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_mutex_setprioceiling

この関数は、引き数 **mutex** によって識別される mutex オブジェクトの優先順位上限属性を動的に設定するために使用できます。新しい上限は、引き数 **new** に入っている値に設定されます。以前の上限は、引き数 **old** によって戻されます。引き数 **new** は、範囲が 1 〜 127 の整数値を想定します。

戻りコード

mutex 優先順位保護プロトコルが AIX にインプリメントされていないため、この関数は現在使用できないことに注意してください。

例

```
integer function f_pthread_mutex_setprioceiling(mutex, new, old)
    type(f_pthread_mutex_t), intent(inout):: mutex
    integer(4), intent(in):: new
    integer(4), intent(out):: old
end function
```

f_pthread_mutex_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介して操作しなければなりません。さらに、このタイプのオブジェクトは Fortran 定数 **PTHREAD_MUTEX_INITIALIZER** を介して静的に初期化できます。

このデータ型は POSIX の **pthread_mutex_t** に相当します。これは、mutex オブジェクトのタイプです。

f_pthread_mutex_trylock

この関数は、mutex オブジェクトの所有権を獲得するために使用できます。 (つまり、この関数はその mutex をロックします。) mutex がすでに他のスレッドによってロックされている場合は、関数はエラー・コード **EBUSY** を戻します。呼び出しスレッドは、さらに処理を実行するために戻りコードを調べることができます。その mutex が呼び出し元自身によってすでにロックされている場合は、再帰的ロックを防止するためにエラーが戻されます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EBUSY
: ターゲットの mutex が他のスレッドによりロックまたは参照されています。

EDEADLK　該当する mutex が、呼び出しスレッドによってすでにロックされています。

EINVAL　引き数 **mutex** が無効です。

例

```
integer function f_pthread_mutex_trylock(mutex)
  type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_mutex_unlock

この関数は、mutex オブジェクトの所有権を解放するためにできるだけ早く呼び出さなくてはなりません。そうすることで、他のスレッドがその mutex をロックできるようになります。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL　引き数 **mutex** が無効です。

EPERM　mutex が、呼び出しスレッドによってロックされていません。

例

```
integer function f_pthread_mutex_unlock(mutex)
    type(f_pthread_mutex_t), intent(inout):: mutex
end function
```

f_pthread_mutexattr_destroy

この関数は、すでに初期化されている mutex 属性オブジェクトを破棄するために使用できます。割り振られたメモリーはその後再収集されます。この属性で作成した mutex は、この処置の影響を受けません。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL　引き数 **mattr** が無効です。

例

```
integer function f_pthread_mutexattr_destroy(mattr)
  type(f_pthread_mutexattr_t), intent(inout):: mattr
end function
```

f_pthread_mutexattr_getprioceiling

この関数は、引き数 **mattr** によって識別される mutex 属性オブジェクト内の mutex 優先順位上限属性を照会するために使用できます。上限属性は引き数 **ceiling** によって戻されます。

戻りコード

mutex 優先順位保護プロトコルが AIX ではインプリメントされていないため、この関数は現在使用できないことに注意してください。

例

```
integer function f_pthread_mutexattr_getprioceiling(mattr, ceiling)
  type(f_pthread_mutexattr_t), intent(in):: mattr
  integer(4), intent(out):: ceiling
end function
```

f_pthread_mutexattr_getprotocol

この関数は、引き数 **mattr** によって識別される mutex 属性オブジェクト内の mutex プロトコル属性の現行設定値を照会するために使用できます。プロトコル属性は引き数 **proto** によって戻されます。

戻りコード

mutex の優先順位継承や優先順位保護が AIX にはインプリメントされていないため、この関数は現在使用できないことに注意してください。

例

```
integer function f_pthread_mutexattr_getprotocol(mattr, proto)
    type(f_pthread_mutexattr_t), intent(in):: mattr
    integer(4), intent(out):: proto
end function
```

f_pthread_mutexattr_getpshared

この関数は、引き数 *mattr* によって識別される mutex 属性オブジェクトのプロセス共用属性を照会するために使用できます。この属性の現行の設定値は、引き数 *pshared* によって戻されます。*pshared* には以下のいずれかの値が入ります。

PTHREAD_PROCESS_SHARED
: mutex は、複数のプロセスによって共用されているメモリーに割り当てられている場合でも、そのメモリーにアクセスしているすべてのスレッドによって操作することができます。

PTHREAD_PROCESS_PRIVATE mutex を操作できるスレッドは、mutex を初期化したスレッドと同じプロセスで作成されたものだけです。

戻りコード

この関数が正常に完了すると、値 0 を戻します。さらに、プロセス共用属性が、引き数 *pshared* を介して戻されます。正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 *mattr* が無効です。

例

```
integer(4) function f_pthread_mutexattr_getpshared(mattr, pshared)
  type(f_pthread_mutexattr_t), intent(in):: mattr
  integer(4), intent(out):: pshared
end function f_pthread_mutexattr_getpshared
```

f_pthread_mutexattr_gettype

この関数は、引き数 **mattr** によって識別される mutex 属性オブジェクトの mutex タイプ属性を照会するために使用できます。

この関数が正常に終了すると、値 0 を戻し、引き数 type を介してタイプ属性を戻します。引き数 type には、次の値の 1 つが含まれます。

PTHREAD_MUTEX_NORMAL このタイプの mutex は、デッドロックを検出することはありません。 この mutex を最初にアンロックせずに再びロックしようとしたスレッドは、デッドロックになります。異なるスレッドによってロックされた mutex をアンロックしようとした場合の動作は未定義です。

PTHREAD_MUTEX_ERRORCHECK
このタイプの mutex は、エラー・チェックを提供します。この mutex を最初にアンロックせずに再びロックしようとしたスレッドは、エラーとともに戻されます。別のスレッドがロックした mutex をアンロックしようとしたスレッドは、エラーを戻します。アンロックをした mutex をアンロックしようとすると、エラーを戻します。

PTHREAD_MUTEX_RECURSIVE この mutex を最初にアンロックせずに再びロックしようとしたスレッドは、mutex のロックに成功します。 タイプ **PTHREAD_MUTEX_NORMAL** の mutex によって起きる可能性がある再ロックによる

デッドロックは、このタイプの mutex では起きません。別のスレッドが mutex を獲得するためには、この mutex をロックしたのと同じ数のアンロックを行って、この mutex を解放しなければなりません。

戻りコード

この関数が失敗すると、以下のエラー・コードを戻します。

EINVAL
引き数が無効です。

例

```
integer(4) function f_pthread_mutexattr_gettype(mattr, type)
  type(f_pthread_mutexattr_t), intent(in):: mattr
  integer(4), intent(out):: type
end function f_pthread_mutexattr_gettype
```

f_pthread_mutexattr_init

この関数を使って mutex 属性オブジェクトを初期化してからでないと、このオブジェクトを別の方法で使用できません。 mutex 属性オブジェクトは引き数 **mattr** によって戻されます。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL 引き数 **mattr** が無効です。

ENOMEM このオブジェクトを作成するにはメモリーが不十分です。

例

```
integer function f_pthread_mutexattr_init(mattr)
    type(f_pthread_mutexattr_t), intent(out):: mattr
end function
```

f_pthread_mutexattr_setprioceiling

この関数は、引き数 **mattr** によって識別される mutex 属性オブジェクト内の mutex 優先順位上限属性を設定するために使用できます。引き数 **ceiling** は、範囲が 1 ～ 127 の整数です。この属性は、mutex 優先順位保護プロトコルが使用される場合にのみ影響します。

戻りコード

優先順位保護プロトコルが AIX ではインプリメントされていないため、この関数は現在使用できないことに注意してください。

例

```
integer function f_pthread_mutexattr_setprioceiling(mattr, ceiling)
  type(f_pthread_mutexattr_t), intent(inout):: mattr
  integer(4), intent(in):: ceiling
end function
```

f_pthread_mutexattr_setprotocol

この関数は、引き数 **mattr** によって識別される mutex 属性オブジェクト内の mutex プロトコル属性を設定するために使用できます。引き数 **proto** は、設定する mutex プロトコルを識別します。 **proto** の有効な値の集合について詳しくは、AIX オペレーティング・システムの資料を参照してください。

戻りコード

mutex の優先順位継承や優先順位保護が AIX にはインプリメントされていないため、この関数は現在使用できないことに注意してください。

例

```
integer function f_pthread_mutexattr_setprotocol(mattr, proto)
    type(p_thread_mutexattr_t), intent(inout):: mattr
    integer(4), intent(in):: proto
end function
```

f_pthread_mutexattr_setpshared

この関数は、引き数 *mattr* によって識別される mutex 属性オブジェクトのプロセス共用属性を設定するために使用できます。引き数 *pshared* には、以下のいずれかの値が入っていなければなりません。

PTHREAD_PROCESS_SHARED mutex が、複数のプロセスによって共用されているメモリーに割り当てられている場合でも、そのメモリーにアクセスしているすべてのスレッドによってその mutex を操作することができるように指定します。

PTHREAD_PROCESS_PRIVATE mutex を操作できるスレッドは、 mutex を初期化したスレッドと同じプロセス内で作成されたスレッドだけとするように指定します。これは、属性のデフォルト設定です。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のエラー・コードを戻します。

EINVAL
引き数が無効です。

例

```
integer(4) function f_pthread_mutexattr_setpshared(mattr, pshared)
  type(f_pthread_mutexattr_t), intent(inout):: mattr
  integer(4), intent(in):: pshared
end function f_pthread_mutexattr_setpshared
```

f_pthread_mutexattr_settype

この関数は、引き数 **mattr** によって識別される mutex 属性オブジェクト内に mutex タイプ属性を設定するために使用できます。引き数 type は、設定する mutex タイプ属性を識別します。 mutex のタイプの詳細については、AIX オペレーティング・システム資料を参照してください。

引き数タイプには、以下のいずれかの値が入っていなければなりません。

PTHREAD_MUTEX_NORMAL
このタイプの mutex は、デッドロックを検出することはありません。 この mutex を最初にアンロックせずに再びロックしようとしたスレッドは、デッドロックになります。異なるスレッドによってロックされた mutex をアンロックしようとした場合の動作は未定義です。

PTHREAD_MUTEX_ERRORCHECK
このタイプの mutex は、エラー・チェックを提供します。この mutex を最初にアンロックせずに再びロックしようとしたスレッドは、エラーとともに戻されます。別のスレッドがロックした mutex をアンロックしようとしたスレッドは、エラーを戻します。アンロックをした mutex をアンロックしようとすると、エラーを戻します。

PTHREAD_MUTEX_RECURSIVE
この mutex を最初にアンロックせずに再びロックしようとしたスレッドは、mutex のロックに成功します。 タイプ **PTHREAD_MUTEX_NORMAL** の mutex によって起きる可能性がある再ロックによるデッドロックは、このタイプの mutex では起きません。別のスレッドが mutex を獲得するためには、こ

の mutex をロックしたのと同じ数のアンロックを行って、この mutex を解放しなければなりません。

PTHREAD_MUTEX_DEFAULT **PTHREAD_MUTEX_NORMAL** と同じです。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のエラー・コードを戻します。

EINVAL
引き数のいずれかが無効です。

例

```
integer(4) function f_pthread_mutexattr_settype(mattr, type)
  type(f_pthread_mutexattr_t), intent(inout):: mattr
  integer(4), intent(in):: type
end function f_pthread_mutexattr_settype
```

f_pthread_mutexattr_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければなりません。

このデータ型は POSIX の **pthread_mutexattr_t** に相当します。これは、mutex 属性オブジェクトのタイプです。

f_pthread_once

この関数は、初期化する必要のあるデータを 1 回だけ初期化するために使用できます。この関数を呼び出す最初のスレッドは、**initr** を呼び出して初期化を実行します。他のスレッドがこの関数を後から呼び出しても、何の影響もありません。引き数 **initr** は、仮引き数のないサブルーチンでなければなりません。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **once** または **initr** が無効です。

例

```
integer function f_pthread_once(once, initr)
    type(f_pthread_once_t), intent(inout):: once
    external initr
end function
```

f_pthread_once_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介して操作しなければなりません。ただし、このタイプのオブジェクトは Fortran 定数 **PTHREAD_ONCE_INIT** によって初期化することだけが可能です。

このデータ型は POSIX の **pthread_once_t** に相当します。これは、once-block オブジェクトのタイプです。

f_pthread_rwlock_destroy

この関数は、引き数 **rwlock** によって指定された読み取り / 書き込みロック・オブジェクトを破棄し、そのロックによって使用されていたすべてのリソースを解放します。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のいずれかのエラー・コードを戻します。

EBUSY
: ターゲットの読み取り / 書き込みオブジェクトがロックされています。

EINVAL
: 引き数 **rwlock** が無効です。

例

```
integer(4) function f_pthread_rwlock_destroy(rwlock)
  type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_destroy
```

f_pthread_rwlock_init

この関数は、**rwlock** に指定された読み取り / 書き込みロック・オブジェクトを、**rwattr** に指定された属性を使用して初期化します。オプションの引き数 **rwattr** が指定されていないと、システムはデフォルトの属性を持つ読み取り / 書き込みロック・オブジェクトを初期化します。初期化後に、ロックは重要なデータへのアクセスを同期化するために使用できます。読み取り / 書き込みロックを使用すると、多くのスレッドが同時にデータへの読み取り専用アクセスを行うことができますが、同時に書き込みアクセスが行えるスレッドは 1 つだけであり、その間他の書き込み機能および読み取り機能は使用できません。スレッドの同期化や読み取り / 書き込みロック・オブジェクトの使用方法の詳細については、AIX オペレーティング・システム資料を参照してください。

読み取り / 書き込みロック・オブジェクトを初期化するための別の方法は、それらを、Fortran 定数 **PTHREAD_RWLOCK_INITIALIZER** によって、静的に初期化するもので

す。この初期化の方法を使用すると、読み取り / 書き込みロック・オブジェクトを使用する前にこの関数を呼び出す必要はありません。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のいずれかのエラー・コードを戻します。

EAGAIN
この読み取り/書き込みロックを初期化するための十分なリソースがシステムにありませんでした。

ENOMEM
この読み取り/ 書き込みロックを初期するための十分なメモリーがありません。

EBUSY
この読み取り/書き込みロックはすでに使用中です。これは初期化されており、破棄されていませんでした。

EINVAL
引き数 rwlock または rwattr が無効です。

EPERM
呼び出し側が操作を行う特権を持っていません。

例

```
integer(4) function f_pthread_rwlock_init(rwlock, rwattr)
  type(f_pthread_rwlock_t), intent(out):: rwlock
  type(f_pthread_rwlockattr_t), intent(in), optional:: rwattr
end function f_pthread_rwlock_init
```

f_pthread_rwlock_rdlock

この関数は、引き数 **rwlock** によって指定された読み取り / 書き込みロックに読み取りロックを適用します。書き込み機能がロックを保留しておらず、ロックに対してブロックされた書き込みがない場合は、呼び出しスレッドは、読み取りロックを獲得します。それ以外の場合、呼び出しスレッドは読み取りロックを獲得しません。 読み取りロックが獲得されない場合、呼び出しスレッドは、ロックを獲得できるまでブロックします (つまり、**f_pthread_rwlock_rdlock** 呼び出しから戻りません)。呼び出しが行われたときに、この呼び出しスレッドが書き込みロックを **rwlock** に対して保留している場合の結果は未定義です。スレッドは複数の並列の読み取りロックを **rwlock** に対して保留にすることがあります (つまり、**f_pthread_rwlock_rdlock** 関数を *n* 回呼び出すことに成功する)。その場合、スレッドは、アンロックのマッチングを実行しなければなりません (つまり、**f_pthread_rwlock_unlock** 関数を *n* 回呼び出さなければならない)。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のいずれかのエラー・コードを戻します。

EAGAIN
: rwlock の読み取りロックが最大数を超えているため、読み取り/書き込みロックを取得できませんでした。

EINVAL
: 引き数 rwlock が、初期化済みの読み取り/書き込みロック・オブジェクトを参照していません。

例

```
integer(4) function f_pthread_rwlock_rdlock(rwlock)
  type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_rdlock
```

f_pthread_rwlock_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければばなりません。さらに、このタイプのオブジェクトは Fortran 定数 **PTHREAD_RWLOCK_INITIALIZER** を介して静的に初期化できます。

このデータ型は、AIX のデータ型 **pthread_rwlock_t** に相当します。これは、読み取り/書き込みロック・オブジェクトのタイプです。

f_pthread_rwlock_tryrdlock

この関数は、**f_pthread_rwlock_rdlock** 関数と同様に読み取りロックを適用します。スレッドが **rwlock** に対して書き込みロックを保留しているか、または **rwlock** に対して書き込み機能がブロックされている場合に、関数が失敗するという点で異なります。そのような場合、関数は **EBUSY** を戻します。呼び出しスレッドはさらに処理を実行するために、戻りコードを調べます。

戻りコード

rwlock によって指定された読み取り / 書き込みロック・オブジェクトの書き込みのロックが獲得されている場合、この関数はゼロを戻します。そうでない場合には、以下のいずれかのエラー・コードを戻します。

EAGAIN
: rwlock の読み取りロックが最大数を超えているため、読み取り/書き込みロックを取得できませんでした。

EBUSY
書き込み機能がロックを保留しているか、ブロックされたため、読み取りのために読み取り / 書き込みロックを獲得できませんでした。

EDEADLK
現在のスレッドは、書き込みのための読み取り/書き込みロックをすでに所有しています。

EINVAL
引き数 rwlock が、初期化済みの読み取り/書き込みロック・オブジェクトを参照していません。

例

```
integer(4) function f_pthread_rwlock_tryrdlock(rwlock)
  type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_tryrdlock
```

f_pthread_rwlock_trywrlock

この関数は、**f_pthread_rwlock_wrlock** 関数と同様に書き込みロックを適用します。スレッドが現在 **rwlock** (読み取りおよび書き込み用) を保留にしている場合に、この関数は失敗するという点が異なります。そのような場合、関数は **EBUSY** を戻します。呼び出しスレッドはさらに処理を実行するために、戻りコードを調べます。

戻りコード

rwlock によって指定された読み取り / 書き込みロック・オブジェクトの書き込みのロックが獲得されている場合、この関数はゼロを戻します。そうでない場合には、以下のいずれかのエラー・コードを戻します。

EBUSY
書き込み機能がロックを保留しているか、ブロックされたため、読み取りのために読み取り / 書き込みロックを獲得できませんでした。

EDEADLK
現在のスレッドは、書き込みのための読み取り/書き込みロックをすでに所有しています。

EINVAL
引き数 rwlock が、初期化済みの読み取り/書き込みロック・オブジェクトを参照していません。

例

```
integer(4) function f_pthread_rwlock_trywrlock(rwlock)
  type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_trywrlock
```

f_pthread_rwlock_unlock

この関数は、引き数 *rwlock* によって指定された読み取り / 書き込みロック・オブジェクト上で保留になっているロックを解放するときに使用します。読み取り / 書き込みロック・オブジェクトから読み取りロックを解放するためにこの関数を呼び出し、さらに現在、この読み取り / 書き込みロック・オブジェクトに他の読み取りロックが存在する場合、読み取り / 書き込みロック・オブジェクトは、読み取りロック状態のままになります。この関数が、読み取り / 書き込みロック・オブジェクト上の呼び出しスレッドの最後の読み取りロックを解放すると、その呼び出しスレッドは、もはやオブジェクトの所有者ではなくなります。この関数が、この読み取り / 書き込みロック・オブジェクト上の最後の読み取りロックを解放すると、読み取り / 書き込みロック・オブジェクトは、所有者がいないアンロックされた状態になります。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のいずれかのエラー・コードを戻します。

EINVAL
: 引き数 **rwlock** が、初期化済みの読み取り / 書き込みロック・オブジェクトを参照していません。

EPERM
: 現在のスレッドは読み取り / 書き込みロックを所有していません。

例

```
integer(4) function f_pthread_rwlock_unlock(rwlock)
  type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_unlock
```

f_pthread_rwlock_wrlock

この関数は、引き数 *rwlock* によって指定された読み取り / 書き込みロックに書き込みロックを適用します。他のスレッド (読み取り機能または書き込み機能) が読み取り / 書き込みロック *rwlock* を保持していない場合、呼び出しスレッドは、書き込みロックを獲得します。書き込みロックがすでに獲得されている場合、そのスレッドは、ロックを獲得するまでブロックされます (つまり、**f_pthread_rwlock_wrlock** 呼び出しから戻らない)。呼び出しが行われたときに、呼び出しスレッド自身が読み取り / 書き込みロック (読み取りロックまたは書き込みロックのどちらか) を保持している場合の結果は未定義です。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 rwlock が、初期化済みの読み取り/書き込みロック・オブジェクトを参照していません。

例

```
integer(4) function f_pthread_rwlock_wrlock(rwlock)
  type(f_pthread_rwlock_t), intent(inout):: rwlock
end function f_pthread_rwlock_wrlock
```

f_pthread_rwlockattr_destroy

この関数は、以前に初期化された引き数 **rwattr** によって指定された読み取り/書き込みロック属性オブジェクトを破棄します。この属性で作成した読み取り / 書き込みロックは、この処置の影響を受けません。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 **rwattr** が無効です。

例

```
integer(4) function f_pthread_rwlockattr_destroy(rwattr)
  type(f_pthread_rwlockattr_t), intent(inout):: rwattr
end function f_pthread_rwlockattr_destroy
```

f_pthread_rwlockattr_getpshared

この関数は、引き数 **rwattr** によって指定された、初期化済み読み取り / 書き込みロック属性オブジェクトから、プロセス共用属性の値を入手するために使用されます。この属性の現行の設定値は、引き数 **pshared** に戻されます。 **pshared** には以下のいずれかの値が入ります。

PTHREAD_PROCESS_SHARED
読み取り / 書き込みロックは、異なるプロセスに属するスレッドがメモリーに割り当てられている場合でも、メモリーにアクセスしているすべてのスレッドによって使用することができます。

PTHREAD_PROCESS_PRIVATE
読み取り / 書き込みロックが使用されるのは、プロセスを作成したスレッドと同じプロセス内でのみであることを指定します。

戻りコード

この関数が正常に完了すると、値 0 が戻され、さらに、*rwattr* のプロセス共用属性が、引き数 *pshared* によって指定されたオブジェクトに格納されます。正常に完了しない場合は、次のエラーを戻します。

EINVAL
引き数 *rwattr* が無効です。

例

```
integer(4) function f_pthread_rwlockattr_getpshared(rwattr, pshared)
  type(f_pthread_rwlockattr_t), intent(in):: rwattr
integer(4), intent(out):: pshared
end function f_pthread_rwlockattr_getpshared
```

f_pthread_rwlockattr_init

この関数は、*rwattr* によって指定された読み取り / 書き込みロック属性を、すべてデフォルト値の属性に初期化します。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 正常に完了しない場合は、次のエラーを戻します。

ENOMEM
読み取り/書き込み属性オブジェクトを初期化するための十分なメモリーがありません。

例

```
integer(4) function f_pthread_rwlockattr_init(rwattr)
  type(f_pthread_rwlockattr_t), intent(out):: rwattr
end function f_pthread_rwlockattr_init
```

f_pthread_rwlockattr_setpshared

この関数は、引き数 *rwattr* によって指定される、初期化された読み取り / 書き込みロック属性オブジェクトにプロセス共用属性を設定するために使用できます。引き数 *pshared* には、以下のいずれかの値が入っていなければなりません。

PTHREAD_PROCESS_SHARED 読み取り / 書き込みロックは、異なるプロセスに属するスレッドがメモリーに割り当てられている場合でも、メモリーにアクセスしているすべてのスレッドによって使用することができます。

PTHREAD_PROCESS_PRIVATE 読み取り / 書き込みロックが使用されるのは、プロ

セスを作成したスレッドと同じプロセス内でのみであることを指定します。これは、属性のデフォルト設定です。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のエラー・コードを戻します。

EINVAL
引き数 **rwattr** が無効です。

例

```
integer(4) function f_pthread_rwlockattr_setpshared(rwattr, pshared)
  type(f_pthread_rwlockattr_t), intent(inout):: rwattr
  integer(4), intent(in):: pshared
end function f_pthread_rwlockattr_setpshared
```

f_pthread_rwlockattr_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければなりません。

このデータ型は、データ型 **pthread_rwlockattr_t** に対応します。これは、読み取り/書き込みロック属性オブジェクトのタイプです。

f_pthread_self

この関数は、呼び出しスレッドのスレッド ID を戻すために使用できます。

戻りコード

呼び出しスレッドの ID が戻されます。

例

```
type(f_pthread_t) function f_pthread_self()
end function
```

f_pthread_setcancelstate

この関数は、スレッドの取り消し可能状態を設定するために使用できます。新しい状態は、引き数 **state** に従って設定されます。以前の状態は、引き数 **oldstate** に戻されます。これらの引き数は、以下のいずれかの Fortran 定数の値を想定します。

PTHREAD_CANCEL_DISABLE: スレッドを取り消しにすることはできません。

PTHREAD_CANCEL_ENABLE: スレッドを取り消しにすることができます。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **state** が無効です。

例

```
integer function f_pthread_setcancelstate(state, oldstate)
  integer(4), intent(in):: state
  integer(4), intent(out):: oldstate
end function
```

f_pthread_setcanceltype

この関数は、スレッドの取り消し可能タイプを設定するために使用できます。新しいタイプは、引き数 **type** に従って設定されます。以前のタイプは、引き数 **oldtype** に戻されます。これらの引き数は、以下のいずれかの Fortran 定数の値を想定します。

PTHREAD_CANCEL_DEFERRED:
取り消し要求は、取り消し点まで実行されません。

PTHREAD_CANCEL_ASYNCHRONOUS:
取り消し要求は、即時実行されます。これは予期しない結果になることがあります。

戻りコード

この関数の実行時にエラーを検出すると、以下のエラー・コードを戻します。

EINVAL 引き数 **type** が無効です。

例

```
integer function f_pthread_setcanceltype(type, oldtype)
  integer(4), intent(in):: type
  integer(4), intent(out):: oldtype
end function
```

f_pthread_setconcurrency

この関数は、pthreads ライブラリー・インストール・システムに、引き数 *new_level* によって指定された望ましい並行性レベルを通知するために使用します。この関数呼び出しの結果として、インストール・システムによって提供される実際の並行性レベルは、未指定になります。並行性レベルの詳細については、 AIX オペレーティング・システム資料を参照してください。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 そうでない場合には、以下のいずれかのエラー・コードを戻します。

EAGAIN
: new_level によって指定された値により、システム・リソースが足りなくなる可能性があります。

EINVAL
: new_level によって指定された値が負です。

例

```
integer(4) function f_pthread_setconcurrency(new_level)
integer(4), intent(in):: new_level
end function f_pthread_setconcurrency
```

f_pthread_setschedparam

この関数は、スレッドのスケジューリング・ポリシーとスケジューリング特性を動的に設定するために使用できます。ターゲットのスレッドは、引き数 **thread** によって識別されます。ターゲットのスレッドの新しいスケジューリング・ポリシーは、引き数 **policy** によって指定します。 AIX における有効なスケジューリング・ポリシーについては、AIX オペレーティング・システムの資料を参照してください。ターゲットのスレッドの新しいスケジューリング特性は、引き数 **param** に指定された値に設定されます。**param** 内の sched_priority フィールドは、スケジューリング優先順位を定義します。その範囲は 1 〜 127 です。

新しいポリシーは、呼び出し元がルート権限を持っていない限り、先入れ先出し (FIFO) またはラウンドロビンに設定できません。新しいスケジューリング特性がターゲットのスレッドに対していつ影響するかの詳細については、AIX オペレーティング・システムの資料を参照してください。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL	引き数 **thread** または **param** が無効です。
ENOSYS	POSIX 優先順位スケジューリング・オプションは AIX ではインプリメントされていません。
ENOTSUP	引き数 **policy** または **param** の値はサポートされていません。
EPERM	ターゲットのスレッドがこの操作を実行することが許可されていないか、あるいはすでに mutex プロトコルになっています。
ESRCH	ターゲット・スレッドが存在しないか、無効です。

例

```
integer function f_pthread_setschedparam(thread, policy, param)
    type(f_pthread_t), intent(inout):: thread
    integer(4), intent(in):: policy
    type(f_sched_param), intent(in):: param
end function
```

f_pthread_setspecific

この関数は、引き数 **key** によって識別されるキーに関連した、呼び出しスレッドのスレッド固有データを設定するのに使用できます。引き数 **arg** (オプション) は、設定するスレッド固有データを識別します。 **arg** が指定されていないと、スレッド固有データは NULL に設定されます。これは、各スレッドごとの初期値です。**arg** 引き数として渡せるのは整数ポインターだけです。 **arg** が整数ポインターでないと、その結果は不定です。

実引き数 **arg** は変数でなければならず、代入ステートメントの左に代入できなければなりません。ベクトル添え字を持つ配列セクションを引き数 **arg** に渡すと、結果は予測不能です。

戻りコード

この関数の実行中にエラーを検出すると、以下のいずれかのエラー・コードを戻します。

EINVAL 引き数 **key** が無効です。

ENOMEM データをキーに関連させるにはメモリーが不十分です。

例

```
integer function f_pthread_setspecific(key, arg)
  type(f_pthread_key_t), intent(in):: key
  pointer(arg, byte)
  optional arg
  intent(in) arg
end function
```

f_pthread_t

コンポーネントがすべてプライベートである派生データ型。このタイプのオブジェクトは、いずれもこのモジュールに用意されている該当のインターフェースを介してのみ操作しなければなりません。

このデータ型は POSIX の **pthread_t** に相当します。これは、スレッド・オブジェクトのタイプです。

f_pthread_testcancel

このサブルーチンは、スレッド内に取り消し点を提供します。これを呼び出すと、保留になっている取り消し要求があり、それが有効な状態になっていれば即時実行されます。

例

```
subroutine f_pthread_testcancel()
end subroutine
```

f_sched_param

このデータ型は、AIX システムのデータ構造 **sched_param** に相当します。これは、システム・データ型です。詳しくは、AIX オペレーティング・システムの資料を参照してください。

例

これは、以下のように定義される公用データ構造です。

```
type f_sched_param
  sequence
  integer sched_priority
end type f_sched_param
```

f_sched_yield

この関数を使用して、呼び出しスレッドが再びそのスレッド・リストの最上部になるまで、そのスレッドがプロセッサーを解放するように強制することができます。

戻りコード

この関数が正常に完了すると、値 0 を戻します。 完了しない場合、値 -1 を戻します。

例

```
integer(4) function f_sched_yield()
end function f_sched_yield
```

f_timespec

これは、AIX システムのデータ構造 **timespec** の Fortran における定義です。 Fortran Pthreads モジュール内では、このタイプのオブジェクトは絶対日時を指定するために使用されます。この*期限絶対日付* は、POSIX 条件変数を待機する場合に使用します。詳しくは、AIX オペレーティング・システムの資料を参照してください。

このデータ構造を使用する 64 ビット・アプリケーションはいずれも、 tv_sec 組み込みデータ型の kind タイプ・パラメーターを指定するために、シンボリック定数 time_size を使用します。これは、32 ビット ・アプリケーションと 64 ビット非 LDT アプリケーションでは「4」、64 ビット LDT アプリケーションでは「8」 に設定されます。

例

これは、以下のように定義されるパブリック・データ構造です。

```
type f_timespec
    sequence
    integer(kind=register_size) tv_nsec
end type f_timespec
```

IBM 拡張 の終り

第 3 部 XL Fortran 言語ユーティリティー

- 浮動小数点制御および照会のプロシージャー
- サービス・プロシージャーおよびユーティリティー・プロシージャー

以下の部では、XL Fortran 言語のその他の特徴について説明しています。

- XL Fortran 言語
- XL Fortran でのマルチスレッド・プログラミング
- ハードウェアと XL Fortran

第 16 章 浮動小数点制御および照会のプロシージャー

XL Fortran には、浮動小数点の状況やプロセッサーの制御レジスターを直接的に照会および制御できる方法がいくつかあります。これには、以下のものが含まれます。

- **fpgets** および **fpsets** サブルーチン
- 浮動小数点制御および照会のための効果的なプロシージャー
- Fortran 2000 ドラフト標準で指定された IEEE 浮動小数点機能

fpgets および **fpsets** サブルーチンは、それぞれ浮動小数点演算の状況の検索と設定を行います。オペレーティング・システム・ルーチンを直接呼び出す代わりに、これらのサブルーチンは **fpstat** という論理値の配列を使用して、情報をあちらこちらに渡します。また、XL Fortran は、浮動小数点状況とプロセッサーの制御レジスターを直接制御できるプロシージャーを `xlf_fp_util` モジュールに備えています。このプロシージャーは **fpgets** および **fpsets** サブルーチンよりも効率的です。このプロシージャーは、浮動小数点および制御レジスターを直接操作するインライン・マシン・インストラクションにマップされます。

XL Fortran は、IEEE 浮動小数点状況セマンティクスの Fortran 2000 ドラフト標準規則を利用するために、**IEEE_ARITHMETIC**、**IEEE_EXCEPTIONS**、および **IEEE_FEATURES** モジュールを提供します。

fpgets fpsets

fpgets および **fpsets** サブルーチンは、それぞれ浮動小数点演算の状況の検索と設定を行います。インクルード・ファイル **fpdc.h** には、これ 2 つのサブルーチンのデータ宣言 (仕様ステートメント) が含まれています。インクルード・ファイル **fpdt.h** はデータ初期化 (データ・ステートメント) を含んでおり、ブロック・データのプログラム単位に含まれていなければなりません。

fpgets は浮動小数点プロセス状況を検索して、**fpstat** という論理配列に結果を格納します。

fpsets は、浮動小数点状況を論理配列 **fpstat** と等しくなるように設定します。

この配列には、浮動小数点丸めモードを指定するのに使用できる論理値が含まれています。 **fpstat** 配列のエレメントに関する例および説明は、「ユーザーズ・ガイド」の『*fpgets* および *fpsets* サブルーチン』を参照してください。

注: XLF_FP_UTIL モジュールは、**fpgets** および **fpsets** サブルーチンよりも効果的な浮動小数点操作の状況を操作するための、いくつかのプロシージャーを提供します。詳細については、『浮動小数点制御および照会のための効果的なプロシージャー』を参照してください。

例

```
CALL fpgets( fpstat )
 ...
CALL fpsets( fpstat )
BLOCK DATA
INCLUDE 'fpdc.h'
INCLUDE 'fpdt.h'
END
```

浮動小数点制御および照会のための効果的なプロシージャー

XL Fortran には、浮動小数点の状況やプロセッサーの制御レジスターを直接的に照会および制御できるプロシージャーがいくつかあります。これらのプロシージャーは浮動小数点の状況および制御レジスター (fpscr) を直接操作するインラインのマシン・インストラクションにマップされるため、 fpgets および fpsets サブルーチンよりも効果的です。

XL Fortran は、モジュール xlf_fp_util を提供します。このモジュールには、これらのプロシージャー用のインターフェースとデータ型定義、およびプロシージャーに必要な名前定数の定義が含まれています。このモジュールにより、リンク時まで待たずに、コンパイル時にこれらのプロシージャーのタイプ・チェックを行うことができます。例にリストされる引き数名は、プロシージャーを呼び出すときにキーワード引き数の名前として使用できます。 xlf_fp_util モジュール用に、次のファイルが提供されています。

ファイル名	ファイル・タイプ	場所
xlf_fp_util.mod	モジュール・シンボル・ファイル (32 ビット)	• /usr/lpp/xlf/include_32_d10 • /usr/lpp/xlf/include_32_d7 **注:** これらのディレクトリー内のファイルは、それぞれまったく同じものです。
	モジュール・シンボル・ファイル (64 ビット)	• /usr/lpp/xlf/include64

これらのプロシージャーを使用するには、ソース・ファイルに USE XLF_FP_UTIL ステートメントを追加する必要があります。 **USE** について、詳しくは 488 ページの『USE』を参照してください。

名前の競合が生じる場合 (たとえば、アクセス元のサブプログラムにモジュール・エンティティーと同じ名前のエンティティーがある場合) は、**ONLY** 節を使用するか、または **USE** ステートメントの名前変更機能を使用してください。以下に例を示します。

```
USE XLF_FP_UTIL, NULL1 => get_fpscr, NULL2 => set_fpscr
```

-U オプションを指定してコンパイルする場合は、これらのプロシージャーの名前をすべて小文字でコーディングする必要があります。このことを忘れないようにするために、ここでは小文字で名前を記します。

fpscr プロシージャーには次のものがあります。
- 845 ページの『clr_fpscr_flags』
- 846 ページの『fp_trap』
- 846 ページの『get_fpscr』
- 847 ページの『get_fpscr_flags』
- 847 ページの『get_round_mode』
- 848 ページの『set_fpscr』
- 848 ページの『set_fpscr_flags』
- 849 ページの『set_round_mode』

次の表は、fpscr で使用される定数をリストしています。

ファミリー	定数	説明
一般	FPSCR_KIND	fpscr フラグ変数用の KIND 型付きパラメーター。
	FP_MODE_KIND	fp_trap 引き数および結果用の KIND 型付きパラメーター。
IEEE 丸めモード	FP_RND_RN	最も近い値への丸め (デフォルト)
	FP_RND_RZ	ゼロ方向への丸め
	FP_RND_RP	正の無限大方向への丸め
	FP_RND_RM	負の無限大方向への丸め
	FP_RND_MODE	fpscr フラグ変数または値から丸めモードを得るために使用される

ファミリー	定数	説明
IEEE 例外使用可能フラグ 1	TRP_INEXACT	inexact トラップを使用可能にする
	TRP_DIV_BY_ZERO	0 除算トラップを使用可能にする
	TRP_UNDERFLOW	アンダーフロー・トラップを使用可能にする
	TRP_OVERFLOW	オーバーフロー・トラップを使用可能にする
	TRP_INVALID	無効トラップを使用可能にする
	FP_ENBL_SUMM	トラップ使用可能の要約またはすべての使用可能化
IEEE 例外状況フラグ	FP_INVALID	無効な演算の例外
	FP_OVERFLOW	オーバーフロー例外
	FP_UNDERFLOW	アンダーフロー例外
	FP_DIV_BY_ZERO	0 除算例外
	FP_INEXACT	Inexact 例外
	FP_ALL_IEEE_XCP	すべての IEEE 例外の要約フラグ
	FP_COMMON_IEEE_XCP	FP_INEXACT 例外を除く、すべての IEEE 例外の要約フラグ
マシン特有の例外の詳細フラグ	FP_INV_SNAN	NaN シグナルを発信
	FP_INV_ISI	無限大 - 無限大
	FP_INV_IDI	無限大 / 無限大
	FP_INV_ZDZ	0 / 0
	FP_INV_IMZ	無限大 * 0
	FP_INV_CMP	非順序比較
	FP_INV_SQRT	負の値の平方根
	FP_INV_CVI	整数への変換エラー
	FP_INV_VXSOFT	ソフトウェアの要求
マシン特有の例外の要約フラグ	FP_ANY_XCP	任意の例外の要約フラグ
	FP_ALL_XCP	すべての例外の要約フラグ
	FP_COMMON_XCP	FP_INEXACT 例外を除く、すべての例外の要約フラグ

ファミリー	定数	説明
fp_trap 定数 **2**	FP_TRAP_SYNC	高精度トラップの使用可能化
	FP_TRAP_OFF	トラップの使用不可
	FP_TRAP_QUERY	照会トラップ・モード
	FP_TRAP_IMP	回復不能な低精度トラップの使用可能化
	FP_TRAP_IMP_REC	回復可能な低精度トラップの使用可能化
	FP_TRAP_FASTMODE	最も速いモードを選択
	FP_TRAP_ERROR	エラー状態
	FP_TRAP_UNIMPL	要求されたモードが利用不能

注:

1. 例外トラップを使用可能にするには、必要な IEEE 例外使用可能フラグを設定する必要があります。
   - その後、fp_trap 呼び出しとともにトラップを生成するようユーザー・プロセスのモードを変更するか、または
   - 適切な **-qflttrap** サブオプションを指定してプログラムをコンパイルする必要があります。コンパイラー・オプション **-qflttrap** およびそのサブオプションについて、詳しくは 「ユーザーズ・ガイド」を参照してください。
2. fp_trap の定数について詳しくは、「*AIX Technical Reference: Base Operating System and Extensions Volume 1*」の fp_trap に関する説明を参照してください。

xlf_fp_util 浮動小数点プロシージャー

| この節では、XLF_FP_UTIL モジュールにある、浮動小数点制御および照会のための効果
| 的なプロシージャーをリストします。

clr_fpscr_flags

clr_fpscr_flags サブルーチンは、MASK 引き数で指定した浮動小数点の状況および制御レジスター・フラグを消去します。 MASK で指定していないフラグには影響がありません。 MASK はタイプ INTEGER(FPSCR_KIND) でなければなりません。 MASK の操作には intrinsic プロシージャーを使用します (542 ページの『整数ビット・モデル』を参照)。

FPSCR 定数について詳しくは、 843 ページの fpscr の定数を参照してください。

例:

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) MASK

MASK=(IOR(FP_OVERFLOW,FP_UNDERFLOW))
CALL clr_fpscr_flags(MASK)
```

clr_fpscr_flags サブルーチンの別の例は、 847 ページの『get_fpscr_flags』 を参照してください。

fp_trap

fp_trap 関数を使用すると、浮動小数点例外がトラップを生成するよう、ユーザー・プロセスのモードを照会または変更できます。引き数 TRAP_MODE は fp_trap 定数でなければなりません。 fp_trap 定数については、 843 ページの fpscr の定数を参照してください。

詳細については、「*AIX Technical Reference: Base Operating System and Extensions Volume 1*」の fp_trap に関する説明を参照してください。

結果タイプおよび属性: fp_trap は、INTEGER(FP_MODE_KIND) を fp_trap 定数の形で戻します。

結果値: 「*AIX Technical Reference: Base Operating System and Extensions Volume 1*」を参照してください。

例:

```
USE XLF_FP_UTIL
INTEGER(FP_MODE_KIND) FP_MODE, TRAP_MODE

TRAP_MODE = FP_TRAP_IMP
FP_MODE = fp_trap(TRAP_MODE)
```

fp_trap の使用方法を示す別の例について、 848 ページの『set_fpscr_flags』 を参照してください。

get_fpscr

get_fpscr 関数は、プロセッサーの浮動小数点状況および制御レジスター (fpscr) の現在の値を戻します。

結果タイプおよび属性: INTEGER(FPSCR_KIND)

結果値: プロセッサーの浮動小数点状況および制御レジスター (fpscr) の現行値。

例:

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR

FPSCR=get_fpscr()
```

get_fpscr_flags

get_fpscr_flags 関数は、MASK 引き数で指定した浮動小数点状況および制御レジスター・フラグの現在の状態を戻します。 MASK はタイプ INTEGER(FPSCR_KIND) でなければなりません。 MASK の操作には intrinsic を使用します (542 ページの『整数ビット・モデル』を参照)。

FPSCR 定数について詳しくは、 843 ページの fpscr の定数を参照してください。

結果タイプおよび属性: INTEGER(FPSCR_KIND)

結果値: FPSCR フラグの状況は MASK 引き数によって指定されます。 MASK 引き数で指定されたフラグがオンになっている場合、そのフラグの値が戻り値に返されます。以下の例は、fp_div_by_zero および fp_invalid フラグの状況を要求します。
- 両方のフラグがオンの場合、戻り値は ior(fp_div_by_zero, fp_invalid) です。
- fp_invalid フラグのみがオンの場合、戻り値は fp_invalid です。
- fp_div_by_zero フラグのみがオンの場合、戻り値は fp_div_by_zero です。
- どちらのフラグもオンになっていない場合、戻り値は 0 です。

例:

```
USE XLF_FP_UTIL

! ...

IF (get_fpscr_flags(IOR(FP_DIV_BY_ZERO,FP_INVALID)) .NE. 0) THEN
  ! Either Divide-by-zero or an invalid operation occurred.

  ! ...

  ! After processing the exception, the exception flags are
  ! cleared.
  CALL clr_fpscr_flags(IOR(FP_DIV_BY_ZERO,FP_INVALID))
END IF
```

get_round_mode

get_round_mode 関数は、現在の浮動小数点丸めモードを戻します。戻り値は、定数 FP_RND_RN、FP_RND_RZ、FP_RND_RP、または FP_RND_RM のいずれか 1 つです。丸めモードの定数について、詳しくは 843 ページの fpscr の定数を参照してください。

結果タイプおよび属性: INTEGER(FPSCR_KIND)

結果値: 定数 FP_RND_RN、FP_RND_RZ、FP_RND_RP、または FP_RND_RM のいずれか。

例:

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) MODE

MODE=get_round_mode()
IF (MODE .EQ. FP_RND_RZ) THEN
! ...
END IF
```

set_fpscr

set_fpscr 関数は、プロセッサーの浮動小数点状況および制御レジスター (fpscr) を FPSCR 引き数で指定された値に設定し、変更を行う前に、そのレジスターの値を戻します。

引き数タイプおよび属性: 整数の kind FPSCR_KIND

結果タイプおよび属性: 整数の kind FPSCR_KIND

結果値: set_fpscr に設定される前のレジスターの値。

例:

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) FPSCR, OLD_FPSCR

FPSCR=get_fpscr()

! ...  Some changes are made to FPSCR  ...

OLD_FPSCR=set_fpscr(FPSCR)  ! OLD_FPSCR is assigned the value of
                            ! the register before it was
                            ! set with set_fpscr
```

set_fpscr_flags

set_fpscr_flags サブルーチンを使用すると、 MASK 引き数で指定した浮動小数点の状況および制御レジスター・フラグを設定できます。 MASK で指定していないフラグには影響がありません。 MASK はタイプ INTEGER(FPSCR_KIND) でなければなりません。 MASK の操作には intrinsic を使用します (542 ページの『整数ビット・モデル』を参照)。

FPSCR 定数について詳しくは、 843 ページの fpscr の定数を参照してください。

例:

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) SAVED_FPSCR
INTEGER(FP_MODE_KIND) FP_MODE

SAVED_FPSCR = get_fpscr()          ! Saves the current value of
                                   ! the fpscr register.
FP_MODE = fp_trap(FP_TRAP_SYNC)    ! Enables precise trapping.

CALL set_fpscr_flags(TRP_DIV_BY_ZERO) ! Enables trapping of
! ...                                 ! divide-by-zero.
FP_MODE=fp_trap(FP_MODE)              ! Restores initial trap
                                      ! mode.
SAVED_FPSCR=set_fpscr(SAVED_FPSCR)    ! Restores fpscr register.
```

set_round_mode

set_round_mode 関数は現在の浮動小数点丸めモードを設定し、変更を行う前に、その丸めモードを戻します。設定できるモードは、FP_RND_RN、FP_RND_RZ、FP_RND_RP、または FP_RND_RM です。丸めモードの定数について、詳しくは 843 ページの fpscr の定数を参照してください。

引き数タイプおよび属性: 整数の kind FPSCR_KIND

結果タイプおよび属性: 整数の kind FPSCR_KIND

結果値: 変更前の丸めモード。

例:

```
USE XLF_FP_UTIL
INTEGER(FPSCR_KIND) MODE

MODE=set_round_mode(FP_RND_RZ)  ! The rounding mode is set to
                                ! round towards zero.  MODE is
! ...                           ! assigned the previous rounding
                                ! mode.
MODE=set_round_mode(MODE)       ! The rounding mode is restored.
```

IEEE モジュールとサポート

IBM 拡張

XL Fortran は、Fortran 2000 のドラフト標準で指定された IEEE 浮動小数点機能のサポートを提供します。ドラフト標準では、例外に対応する **IEEE_EXCEPTIONS** モジュール、 IEEE 演算をサポートする **IEEE_ARITHMETIC** モジュール、およびコンパイラーによってサポートされる IEEE 機能を指定する **IEEE_FEATURES** が定義されています。

IEEE_EXCEPTIONS または **IEEE_ARITHMETIC** 組み込みモジュールを使用する場合、 XL Fortran コンパイラーは、丸めモード、停止モード、および例外フラグについての浮動小数点状況に対する変更の有効範囲に関するいくつかの Fortran 2000 のドラフト標準規則を実施しています。このことで、上記のモジュールを使用していても、新しい浮動小数点状況セマンティクスを使用しないプログラムのパフォーマンスが低下する可能性があります。そのようなプログラムに対して、浮動小数点状況の保管と復元に関する規則を緩和するために、**-qstrictieeemod** コンパイラー・オプションが提供されています。

注:

1. XL Fortran 拡張精度浮動小数点数は、IEEE 標準で提唱されている形式にはなっていません。結果として、モジュールの一部で **REAL(16)** をサポートしません。
2. **IEEE_SET_FLAG** サブルーチンは、POWER および POWER2 プラットフォームで **IEEE_INVALID** 例外フラグを設定しません。
3. IEEE モジュールの停止機能を使用するプログラムは、**-qflttrap** オプションを指定してコンパイルする必要があります。停止を使用可能にしていて例外が発生すると、IEEE モジュールは **SIGTRAP** シグナルを生成します。**-qflttrap=imprecise** コンパイラー・オプションを指定すれば、Fortran 2000 ドラフト標準の要件に準拠しつつ、停止のパフォーマンスへの影響は少なくなります。

コンパイルと例外処理

XL Fortran は、IEEE 標準に完全に準拠するための多くのオプションを提供しています。

- **-qfloat=nomaf** を使用して、浮動小数点数演算の IEEE 標準 (IEEE 754-1985) に準拠するようにします。
- 丸めモードを変更するプログラムをコンパイルするときは、**-qfloat=rrm** を使用します。
- **-qfloat=nans** を使用して、シグナル NaN 値を検出します。シグナル NaN 値は、プログラムで指定されている場合にのみ発生します。
- 最適化レベル **-O3** 以上、**-qhot**、**-qipa**、**-qpdf**、または **-qsmp** でコンパイルしたプログラムで浮動小数点数演算の IEEE 標準に完全に準拠するには、**-qstrict** コンパイラー・オプションを使用します。
- **-qarch=ppc** コンパイラー・オプション、または使用している PowerPC ハードウェア用の該当する **-qarch** サブオプションを指定した代入または **INT** 組み込みを介した、整数変換演算への呼び出しを含むプログラムをコンパイルします。該当する **-qarch** コンパイラー・オプションを指定することで、PowerPC プラットフォームでの無効な整数変換演算をシグナル通知するための **IEEE_INVALID** 例外が可能になります。
- AIX 4.3 以降での無効な **SQRT** 演算をシグナル通知するための **IEEE_INVALID** 例外を可能にするには、次のコマンドを指定します。

```
export SQRT_EXCEPTION=3.1
```

- 無効な整数変換および平方根の演算では、POWER プラットフォーム上で **IEEE_INVALID** 例外フラグを設定しません。

関連情報

IEEE 浮動小数点の詳細、および上記のコンパイラー・オプションの特定の説明については、「ユーザーズ・ガイド」の『*XL Fortran 浮動小数点処理*』を参照してください。

IEEE をインプリメントするための一般規則

IEEE_ARITHMETIC、**IEEE_EXCEPTIONS**、**IEEE_FEATURES** は組み込みモジュールです。ただし、これらのモジュールで定義されるタイプおよびプロシージャーは組み込みではありません。

IEEE モジュールに含まれるすべての関数は純粋 (PURE) です。

すべてのプロシージャー名は汎用であり、特定ではありません。

すべての例外フラグのデフォルト値は静止です。

デフォルトでは、例外は停止を起こさせません。

丸めモードのデフォルトは、最も近い値への丸めとなります。

IEEE 派生データ型と定数

IEEE モジュールは、以下の派生型を定義します。

IEEE_FLAG_TYPE

特定の例外フラグを識別する **IEEE_EXCEPTIONS** モジュールによって定義される派生データ型。**IEEE_FLAG_TYPE** の値は、**IEEE_EXCEPTIONS** モジュールで定義される以下の名前付き定数のいずれかである必要があります。

IEEE_OVERFLOW

組み込みの実数の演算または割り当ての結果に、大きすぎて表すことができない指数があるときに発生します。また、この例外は、組み込みの複素数の演算または割り当ての結果の実数部分または虚数部分に、大きすぎて表すことができない指数があるときにも発生します。

REAL(4) を使用している場合は、結果値の不偏指数が 127 より大か、または -126 より小であるときに、オーバーフローが発生します。

REAL(8) を使用している場合は、結果値の不偏指数が 1023 より大か、または -1022 より小であるときに、オーバーフローが発生します。

IEEE_DIVIDE_BY_ZERO

実数または複素数の除算でゼロ以外の分子とゼロの分母があるときに発生します。

IEEE_INVALID
実数演算または複素数の演算または割り当てが無効であるときに、発生します。

IEEE_UNDERFLOW
組み込みの実数の演算または割り当ての結果に、小さすぎてゼロ以外の値で表すことができない絶対値があり、精度の消失が検出されるときに発生します。また、この例外は、組み込みの複素数の演算または割り当ての結果の実数部分または虚数部分に、小さすぎてゼロ以外の値で表すことができない絶対値があり、精度の消失が検出されるときにも発生します。

REAL(4) を使用している場合は、結果の絶対値が 2^{-149} より小であるときに、アンダーフローが発生します。

REAL(8) を使用している場合は、結果の絶対値が 2^{-1074} より小であるときに、アンダーフローが発生します。

IEEE_INEXACT
実数または複素数の割り当てまたは演算が不正確であるときに、発生します。

以下の定数は、**IEEE_FLAG_TYPE** の配列です。

IEEE_USUAL
順に **IEEE_OVERFLOW**、**IEEE_DIVIDE_BY_ZERO**、および **IEEE_INVALID** のエレメントを含む配列名前付き定数。

IEEE_ALL
順に **IEEE_USUAL**、**IEEE_UNDERFLOW**、および **IEEE_INEXACT** のエレメントを含む配列名前付き定数。

IEEE_STATUS_TYPE

現在の浮動小数点状況を表す、**IEEE_ARITHMETIC** モジュールで定義される派生データ型。浮動小数点状況は、すべての例外フラグ、停止、および丸めモードの値を網羅します。

IEEE_CLASS_TYPE

1 つのクラスの浮動小数点値をカテゴリー化する **IEEE_ARITHMETIC** モジュールで定義される派生データ型。**IEEE_CLASS_TYPE** の値は **IEEE_ARITHMETIC** モジュールで定義される以下の名前付き定数のいずれかである必要があります。

IEEE_SIGNALING_NAN	**IEEE_NEGATIVE_ZERO**
IEEE_QUIET_NAN	**IEEE_POSITIVE_ZERO**
IEEE_NEGATIVE_INF	**IEEE_POSITIVE_DENORMAL**
IEEE_NEGATIVE_NORMAL	**IEEE_POSITIVE_NORMAL**

IEEE_NEGATIVE_DENORMAL	**IEEE_POSITIVE_INF**

IEEE_ROUND_TYPE

特定の丸めモードを識別する **IEEE_ARITHMETIC** モジュールで定義される派生データ型。**IEEE_ROUND_TYPE** の値は、**IEEE_ARITHMETIC** モジュールで定義される以下の名前付き定数のいずれかである必要があります。

IEEE_NEAREST
正確な結果を最も近い表現可能な数に丸めます。

IEEE_TO_ZERO
正確な結果を、ゼロの方向で次の表現可能な数に丸めます。

IEEE_UP
正確な結果を、正の方向で次の表現可能な数に丸めます。

IEEE_DOWN
正確な結果を、負の方向で次の表現可能な数に丸めます。

IEEE_OTHER
丸めモードが IEEE 標準に準拠しないことを示します。

IEEE_FEATURES_TYPE

使用する IEEE 機能を識別する、**IEEE_FEATURES** モジュールで定義される派生データ型。**IEEE_FEATURES_TYPE** の値は、**IEEE_FEATURES** モジュールで定義される以下の名前付き定数のいずれかである必要があります。

IEEE_DATATYPE	**IEEE_DATATYPE**
IEEE_DENORMAL	**IEEE_INVALID_FLAG**
IEEE_DIVIDE	**IEEE_NAN**
IEEE_HALTING	**IEEE_ROUNDING**
IEEE_INEXACT_FLAG	**IEEE_SQRT**
IEEE_INF	**IEEE_UNDERFLOW_FLAG**

IEEE 演算子

IEEE_ARITHMETIC モジュールは、**IEEE_CLASS_TYPE** または **IEEE_ROUND_TYPE** の変数を比較するための 2 組のエレメント型演算子を定義します。

==
2 つの **IEEE_CLASS_TYPE** 値、または 2 つの **IEEE_ROUND_TYPE** 値を比較できるようにします。この演算子は、値が同一である場合は true を戻し、値が異なる場合は false を戻します。

/= 2 つの **IEEE_CLASS_TYPE** 値、または 2 つの **IEEE_ROUND_TYPE** 値を比較できるようにします。この演算子は、値が異なる場合は true を戻し、値が同一である場合は false を戻します。

IEEE プロシージャー

以下の IEEE プロシージャーを使用するには、必要に応じて、`USE IEEE_ARITHMETIC`、`USE IEEE_EXCEPTIONS`、または `USE IEEE_FEATURES` ステートメントをソース・ファイルに追加する必要があります。**USE** ステートメントの詳細については、488 ページの『USE』を参照してください。

IEEE プロシージャーの使用規則

XL Fortran は、**IEEE_FEATURES** モジュールのすべての名前付き定数をサポートします。

IEEE_ARITHMETIC モジュールは、**IEEE_EXCEPTIONS** に対して、**USE** ステートメントを含んでいるときと同様に動作します。**IEEE_EXCEPTIONS** で public であるすべての値は、 **IEEE_ARITHMETIC** でも public のままです。

IEEE_EXCEPTIONS または **IEEE_ARITHMETIC** モジュールがアクセス可能である場合、**IEEE_OVERFLOW** と **IEEE_DIVIDE_BY_ZERO** はすべての種類の実数および複素数のデータの有効範囲単位内でサポートされます。サポートされている他の例外を判別するには、 **IEEE_SUPPORT_FLAG** 関数を使用します。停止がサポートされるかどうかを判別するには、**IEEE_SUPPORT_HALTING** を使用します。他の例外のサポートについては、**IEEE_FEATURES** モジュールの名前付き定数 **IEEE_INEXACT_FLAG**、**IEEE_INVALID_FLAG**、および **IEEE_UNDERFLOW_FLAG** がアクセス可能かどうかによって、以下のような影響を受けます。

- ある有効範囲単位で **IEEE_FEATURES** の **IEEE_UNDERFLOW_FLAG** にアクセスできる場合、その有効範囲単位はアンダーフローをサポートし、**REAL(4)** および **REAL(8)** に対して、 **IEEE_SUPPORT_FLAG(IEEE_UNDERFLOW, X)** から true を戻します。
- **IEEE_INEXACT_FLAG** または **IEEE_INVALID_FLAG** がアクセス可能である場合、有効範囲単位はその例外をサポートし、**REAL(4)** および **REAL(8)** に対して、対応する照会から true を戻します。
- **IEEE_HALTING** がアクセス可能である場合、有効範囲単位は停止制御をサポートし、フラグに対して、**IEEE_SUPPORT_HALTING(FLAG)** を戻します。

IEEE_EXCEPTIONS または **IEEE_ARITHMETIC** にアクセスしない有効範囲単位への入り口で例外フラグがシグナル通知される場合、コンパイラーは、出口で、その例外フラグが確実に通知されるようにします。このような有効範囲単位への入り口でフラグが静止である場合は、出口でフラグがシグナル通知される可能性があります。

これ以上の IEEE サポートは、**IEEE_ARITHMETIC** モジュールを介して取得することができます。サポートは、**IEEE_FEATURES** モジュールの名前付き定数がアクセス可能かどうかによって影響を受けます。

- 有効範囲単位が **IEEE_FEATURES** の **IEEE_DATATYPE** にアクセスできる場合、その有効範囲単位は IEEE 演算をサポートし、**REAL(4)** および **REAL(8)** に対して **IEEE_SUPPORT_DATATYPE(X)** から true を戻します。
- **IEEE_DENORMAL**、**IEEE_DIVIDE**、**IEEE_INF**、**IEEE_NAN**、 **IEEE_ROUNDING**、または **IEEE_SQRT** がアクセス可能である場合、有効範囲単位はその機能をサポートし、**REAL(4)** および **REAL(8)** に対して、対応する照会関数から true を戻します。
- **IEEE_ROUNDING** の場合には、有効範囲単位は、**REAL(4)** および **REAL(8)** に対して、 すべての丸めモード **IEEE_NEAREST**、**IEEE_TO_ZERO**、**IEEE_UP**、および **IEEE_DOWN** に true を戻します。

IEEE_EXCEPTIONS または **IEEE_ARITHMETIC** モジュールにアクセスし、**IEEE_FEATURES** にはアクセスしない場合には、サポートされる機能のサブセットは、 **IEEE_FEATURES** にアクセスした場合と同じです。

IEEE_CLASS(X)

エレメント型 IEEE クラス関数。浮動小数点値の IEEE クラスを戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: 結果は、**IEEE_CLASS_TYPE** タイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** 関数は、 true の値を戻す必要があります。**REAL(16)** のデータ型を指定した場合には、 **IEEE_SUPPORT_DATATYPE** は false を戻します。ただし、クラス・タイプは所定のものが戻されます。

例:

```
USE IEEE_ARITHMETIC
TYPE(IEEE_CLASS_TYPE) :: C
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  C = IEEE_CLASS(X)                ! C has class IEEE_NEGATIVE_NORMAL
ENDIF
```

IEEE_COPY_SIGN(X, Y)

エレメント型 IEEE 符号コピー関数。*Y* の符号を付けた *X* の値を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* および *Y* は実数タイプです。ただし kind は異なる可能性があります。

結果タイプおよび属性: 結果は、*X* と同じ kind およびタイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** および **IEEE_SUPPORT_DATATYPE(Y)** は、 true の値を戻す必要があります。

NaN および無限大など、サポートされている IEEE 特殊値に対して、**IEEE_COPY_SIGN** は *Y* の符号が付いた *X* の値を戻します。

IEEE_COPY_SIGN は、**-qxlf90=nosignedzero** コンパイラー・オプションを無視します。

注: XL Fortran の **REAL(16)** 数値には、符号付きゼロがありません。

例: **例 1:**

```
USE IEEE_ARITHMETIC
REAL :: X
DOUBLE PRECISION :: Y
X =  3.0
Y = -2.0
IF (IEEE_SUPPORT_DATATYPE(X) .AND. IEEE_SUPPORT_DATATYPE(Y)) THEN
  X = IEEE_COPY_SIGN(X,Y)               ! X has value -3.0
ENDIF
```

例 2:

```
USE IEEE_ARITHMETIC
REAL :: X, Y
Y =  1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  X = IEEE_VALUE(X, IEEE_NEGATIVE_INF)  ! X has value -inf
  X = IEEE_COPY_SIGN(X,Y)               ! X has value +inf
ENDIF
```

IEEE_GET_FLAG(FLAG, FLAG_VALUE)

エレメント型 IEEE サブルーチン。指定された、例外フラグの状況を検索します。フラグがシグナル通知である場合に *FLAG_VALUE* を true に設定し、そうでない場合は false に設定します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*FLAG* は、取得する IEEE フラグを指定するタイプ **IEEE_FLAG_TYPE** の **INTENT(IN)** 引き数です。*FLAG_VALUE* は、*FLAG* の値を含む **INTENT(OUT)** デフォルト論理引き数です。

例:

```
USE IEEE_EXCEPTIONS
LOGICAL :: FLAG_VALUE
CALL IEEE_GET_FLAG(IEEE_OVERFLOW,FLAG_VALUE)
IF (FLAG_VALUE) THEN
  PRINT *, "Overflow flag is signaling."
ELSE
  PRINT *, "Overflow flag is quiet."
ENDIF
```

IEEE_GET_HALTING_MODE(FLAG, HALTING)

エレメント型 IEEE サブルーチン。例外に対する停止モードを検索し、フラグによって指定された例外で停止を起こさせる場合は、*HALTING* を true に設定します。**-qflttrap=imprecise** を使用する場合、停止は正確ではなく、例外の後に発生することがあります。デフォルトでは、例外は XL Fortran で停止を起こさせません。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*FLAG* は、IEEE フラグを指定するタイプ **IEEE_FLAG_TYPE** の **INTENT(IN)** 引き数です。*HALTING* は **INTENT(OUT)** デフォルト論理です。

例:

```
USE IEEE_EXCEPTIONS
LOGICAL HALTING
CALL IEEE_GET_HALTING_MODE(IEEE_OVERFLOW,HALTING)
IF (HALTING) THEN
  PRINT *, "The program will halt on an overflow exception."
ENDIF
```

IEEE_GET_ROUNDING_MODE (ROUND_VALUE)

IEEE サブルーチン。*ROUND_VALUE* を現行の IEEE 丸めモードに設定します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*ROUND_VALUE* はタイプ **IEEE_ROUND_TYPE** の **INTENT(OUT)** スカラーです。

例:

```
USE IEEE_ARITHMETIC
TYPE(IEEE_ROUND_TYPE) ROUND_VALUE
CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE) ! Store the rounding mode
IF (ROUND_VALUE == IEEE_OTHER) THEN
  PRINT *, "You are not using an IEEE rounding mode."
ENDIF
```

IEEE_GET_STATUS(STATUS_VALUE)

IEEE サブルーチン。現行の IEEE 浮動小数点状況を検索します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*STATUS_VALUE* はタイプ **IEEE_STATUS_TYPE** の **INTENT(OUT)** スカラーです。

規則: **IEEE_SET_STATUS** 呼び出しでは、*STATUS_VALUE* だけを使用できます。

例:

```
USE IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)   ! Get status of all exception flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set all exception flags to quiet
... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE)   ! Restore the flags
```

IEEE_IS_FINITE(X)

エレメント型 IEEE 関数。値が有限であるかどうかを検査します。**IEEE_CLASS(X)** が以下の値のいずれかを持っている場合には、true を戻します。

- **IEEE_NEGATIVE_NORMAL**
- **IEEE_NEGATIVE_DENORMAL**
- **IEEE_NEGATIVE_ZERO**
- **IEEE_POSITIVE_ZERO**
- **IEEE_POSITIVE_DENORMAL**
- **IEEE_POSITIVE_NORMAL**

それ以外の場合には、false を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: ここで、結果はデフォルト論理タイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

例:

```
USE IEEE_ARITHMETIC
REAL :: X = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, IEEE_IS_FINITE(X)    ! Prints true
ENDIF
```

IEEE_IS_NAN(X)

エレメント型 IEEE 関数。値が IEEE 非数値であるかどうかを検査します。**IEEE_CLASS(X)** が値 **IEEE_SIGNALING_NAN** または **IEEE_QUIET_NAN** を持っている場合には、true を戻します。それ以外の場合には、false を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: ここで、結果はデフォルト論理タイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** および **IEEE_SUPPORT_NAN(X)** は、 true の値を戻す必要があります。

例: **例 1:**

```
USE IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  IF (IEEE_SUPPORT_SQRT(X)) THEN    ! IEEE-compliant SQRT function
    IF (IEEE_SUPPORT_NAN(X)) THEN
      PRINT *, IEEE_IS_NAN(SQRT(X)) ! Prints true
    ENDIF
  ENDIF
ENDIF
```

例 2:

```
USE IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_STANDARD(X)) THEN
  PRINT *, IEEE_IS_NAN(SQRT(X))     ! Prints true
ENDIF
```

IEEE_IS_NEGATIVE(X)

エレメント型 IEEE 関数。値が負であるかどうかを検査します。**IEEE_CLASS(X)** が以下の値のいずれかを持っている場合には、true を戻します。

- **IEEE_NEGATIVE_NORMAL**
- **IEEE_NEGATIVE_DENORMAL**
- **IEEE_NEGATIVE_ZERO**
- **IEEE_NEGATIVE_INF**

それ以外の場合には、false を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: ここで、結果はデフォルト論理タイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

例:

```
USE IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.0)) THEN
  PRINT *, IEEE_IS_NEGATIVE(1.0)    ! Prints false
ENDIF
```

IEEE_IS_NORMAL(X)

エレメント型 IEEE 関数。値が正規であるかどうかを検査します。**IEEE_CLASS(X)** が以下の値のいずれかを持っている場合には、true を戻します。

- **IEEE_NEGATIVE_NORMAL**
- **IEEE_NEGATIVE_ZERO**
- **IEEE_POSITIVE_ZERO**
- **IEEE_POSITIVE_NORMAL**

それ以外の場合には、false を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: ここで、結果はデフォルト論理タイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

例:

```
USE IEEE_ARITHMETIC
REAL :: X = -1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
```

```
  IF (IEEE_SUPPORT_SQRT(X)) THEN     ! IEEE-compliant SQRT function
    PRINT *, IEEE_IS_NORMAL(SQRT(X)) ! Prints false
  ENDIF
ENDIF
```

IEEE_LOGB(X)

エレメント型 IEEE 関数。浮動小数点形式で不偏指数を戻します。*X* の値がゼロでも、無限大でも、NaN でもない場合には、結果は、**EXPONENT(X)-1** と等しい、*X* の不偏指数の値を持ちます。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: ここで、結果は *X* と同じタイプおよび kind です。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X がゼロの場合、結果は負の無限大です。

X が無限大の場合、結果は正の無限大です。

X が NaN の場合、結果は nan です。

例:

```
USE IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
  PRINT *, IEEE_LOGB(1.1)  ! Prints 0.0
ENDIF
```

IEEE_NEXT_AFTER(X, Y)

エレメント型 IEEE 関数。*Y* の方向で、マシンによる表現可能な *X* の次の近隣を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* および *Y* は実数タイプです。

結果タイプおよび属性: ここで、結果は *X* と同じタイプおよび kind です。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** および **IEEE_SUPPORT_DATATYPE(Y)** は、 true の値を戻す必要があります。

X と *Y* が等しい場合、関数は、例外をシグナル通知せずに *X* を戻します。*X* と *Y* が等しくない場合には、*Y* の方向で、マシンによる表現可能な *X* の次の近隣を戻します。

いずれかの符号のゼロの近隣は、両方とも非ゼロです。

X は有限だが、**IEEE_NEXT_AFTER(X, Y)** が無限である場合には、**IEEE_OVERFLOW** および **IEEE_INEXACT** がシグナル通知されます。

IEEE_NEXT_AFTER(X, Y) が非正規またはゼロである場合には、 **IEEE_UNDERFLOW** と **IEEE_INEXACT** がシグナル通知されます。

X または *Y* が静止 NaN である場合には、結果は入力 NaN 値の 1 つです。

例: **例 1:**

```
USE IEEE_ARITHMETIC
REAL :: X = 1.0, Y = 2.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, (IEEE_NEXT_AFTER(X,Y) == X + EPSILON(X))  ! Prints true
ENDIF
```

例 2:

```
USE IEEE_ARITHMETIC
REAL(4) :: X = 0.0, Y = 1.0
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
  PRINT *, (IEEE_NEXT_AFTER(X,Y) == 2.0**(-149))  ! Prints true
ENDIF
```

IEEE_REM(X, Y)

エレメント型 IEEE 剰余関数。結果値は、丸めモードにかかわりなく、正確に *X-Y*N* です。ただし、*N* は *X/Y* の正確な値に最も近い表現可能な整数で、|*N* - *X/Y*| = 1/2 の場合は、常に *N* は偶数です。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* および *Y* は実数タイプです。

結果タイプおよび属性: ここで、結果は、より精度の高い引き数と同じ kind を持つ実数タイプです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** および **IEEE_SUPPORT_DATATYPE(Y)** は、 true の値を戻す必要があります。

結果値がゼロである場合には、符号は *X* と同じです。

例:

```
USE IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(4.0)) THEN
  PRINT *, IEEE_REM(4.0,3.0)  ! Prints  1.0
  PRINT *, IEEE_REM(3.0,2.0)  ! Prints -1.0
  PRINT *, IEEE_REM(5.0,2.0)  ! Prints  1.0
ENDIF
```

IEEE_RINT(X)

エレメント型 IEEE 関数。現行の丸めモードに従って、整数値に丸めます。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。

結果タイプおよび属性: ここで、結果は *X* と同じタイプおよび kind です。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

結果が値ゼロを持つ場合には、符号は *X* と同じです。

例:

```
USE IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
  CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
  PRINT *, IEEE_RINT(1.1)       ! Prints 1.0
  CALL IEEE_SET_ROUNDING_MODE(IEEE_UP)
  PRINT *, IEEE_RINT(1.1)       ! Prints 2.0
ENDIF
```

IEEE_SCALB(X, I)

エレメント型 IEEE 関数。$X * 2^I$ を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプで、*I* は **INTEGER** タイプです。

結果タイプおよび属性: ここで、結果は *X* と同じタイプおよび kind です。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

$X * 2^I$ が正規値として表せる場合には、結果は正規値です。

X が有限で、$X * 2^I$ が大きすぎる場合には、 **IEEE_OVERFLOW** 例外が発生します。結果値は、*X* の符号が付いた無限大になります。

$X * 2^I$ が小さすぎて、精度の喪失がある場合には、 **IEEE_UNDERFLOW** 例外が発生します。結果は、 *X* の符号が付いた最も近い表現可能な数になります。

X が無限大の場合、結果は、例外がシグナル通知されない *X* と同じです。

例:

```
USE IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.0)) THEN
  PRINT *, IEEE_SCALB(1.0,2)       ! Prints 4.0
ENDIF
```

IEEE_SELECTED_REAL_KIND([P, R])

変形 IEEE 関数。少なくとも *P* 桁で、少なくとも *R* の小数部指数範囲を持つ IEEE 実数データ型の kind 型付きパラメーターの値を戻します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*P* および *R* は両方ともオプションの整数タイプのスカラー引き数です。

規則: kind 型付きパラメーターが使用不能で、精度が使用不能である場合には、結果は -1 です。kind 型付きパラメーターが使用不能で、指数範囲が使用不能である場合には、結果は -2 です。kind 型付きパラメーターが使用不能で、精度と指数範囲のいずれも使用不能である場合には、結果は -3 です。

複数の kind 型付きパラメーター値が適用できる場合には、戻される値は最小の小数部精度を持つ値になります。値がいくつかある場合には、それらの kind 値のうち最小のものが戻されます。

例:

```
USE IEEE_ARITHMETIC

! P and R fit in a real(4)
PRINT *, IEEE_SELECTED_REAL_KIND(6,37)   ! prints 4

! P needs at least a real(8)
PRINT *, IEEE_SELECTED_REAL_KIND(14,37)  ! prints 8
! R needs at least a real(8)
PRINT *, IEEE_SELECTED_REAL_KIND(6,307)  ! prints 8

! P is too large
PRINT *, IEEE_SELECTED_REAL_KIND(40,37)  ! prints -1
! R is too large
PRINT *, IEEE_SELECTED_REAL_KIND(6,400)  ! prints -2
```

```
! P and R are both too large
PRINT *, IEEE_SELECTED_REAL_KIND(40,400) ! prints -3

END
```

IEEE_SET_FLAG(FLAG, FLAG_VALUE)

IEEE サブルーチン。IEEE 例外フラグに値を割り当てます。

モジュール: **IEEE_EXCEPTIONS**

構文: ここで、*FLAG* は、設定されるフラグの値に対応するタイプ **IEEE_FLAG_TYPE** の **INTENT(IN)** スカラーまたは配列引き数です。 *FLAG_VALUE* は、例外フラグの望ましい状況に対応する、論理タイプの **INTENT(IN)** スカラーまたは配列引き数です。 *FLAG_VALUE* の値は *FLAG* の値と整合している必要があります。

規則: *FLAG_VALUE* が true である場合、*FLAG* で指定する例外フラグはシグナル通知に設定されます。それ以外の場合には、フラグは静止に設定されます。

FLAG の各エレメントは固有の値を持たなければなりません。

例:

```
USE IEEE_EXCEPTIONS
CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)
! IEEE_OVERFLOW is now signaling
```

IEEE_SET_HALTING_MODE(FLAG, HALTING)

IEEE サブルーチン。例外の後の継続または停止を制御します。

モジュール: **IEEE_EXCEPTIONS**

構文: ここで、*FLAG* は、保留を適用する例外フラグに対応するタイプ **IEEE_FLAG_TYPE** の **INTENT(IN)** スカラーまたは配列引き数です。*HALTING* は、希望する停止状況に対応する、論理タイプの **INTENT(IN)** スカラーまたは配列引き数です。デフォルトでは、例外は XL Fortran で停止を起こさせません。 *HALTING* の値は *FLAG* の値と整合している必要があります。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

-qflttrap=imprecise コンパイラー・オプションを使用する場合、停止は正確ではなく、例外の後に発生することがあります。

HALTING が true の場合、*FLAG* によって指定された例外は停止を発生させます。それ以外の場合は、実行は例外後も継続します。

例外フラグに対してコーディングで停止モードを設定し、 **-qflttrap=enable** オプションを使用しないでプログラム全体をコンパイルすると、例外発生時にプログラムは予期しない結果を発生させます。詳細については、「ユーザーズ・ガイド」を参照してください。

FLAG の各エレメントは固有の値を持たなければなりません。

例:

```
@PROCESS FLOAT(NOFOLD)
USE IEEE_EXCEPTIONS
REAL :: X
CALL IEEE_SET_HALTING_MODE(IEEE_DIVIDE_BY_ZERO, .TRUE.)
X = 1.0 / 0.0
! Program will halt with a divide-by-zero exception
```

IEEE_SET_ROUNDING_MODE (ROUND_VALUE)

IEEE サブルーチン。現行の丸めモードを設定します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*ROUND_VALUE* は、丸めモードを指定する **IEEE_ROUND_TYPE** タイプの **INTENT(IN)** 引き数です。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** および **IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)** は、true の値を戻す必要があります。

このプログラムを呼び出すコンパイル単位は、**-qfloat=rrm** コンパイラー・オプションでコンパイルする必要があります。

-qfloat=rrm コンパイラー・オプションでコンパイルされたプログラムを呼び出すすべてのコンパイル単位も、このオプションでコンパイルする必要があります。

例:

```
USE IEEE_ARITHMETIC
IF (IEEE_SUPPORT_DATATYPE(1.1)) THEN
  CALL IEEE_SET_ROUNDING_MODE(IEEE_NEAREST)
  PRINT *, IEEE_RINT(1.1)       ! Prints 1.0
  CALL IEEE_SET_ROUNDING_MODE(IEEE_UP)
  PRINT *, IEEE_RINT(1.1)       ! Prints 2.0
ENDIF
```

IEEE_SET_STATUS(STATUS_VALUE)

IEEE サブルーチン。浮動小数点状況の値を復元します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*STATUS_VALUE* は、浮動小数点状況を指定する **IEEE_STATUS_TYPE** タイプの **INTENT(IN)** 引き数です。

規則: *STATUS_VALUE* は **IEEE_GET_STATUS** で事前に設定されていなければいけません。

IEEE_SUPPORT_DATATYPE または IEEE_SUPPORT_DATATYPE(X)

照会 IEEE 関数。現行のインプリメンテーションが IEEE 演算をサポートするかどうかを判別します。サポートするとは、オペランドと結果がすべて正規値を持つ場合は必ず、 IEEE 標準に従って、IEEE データ形式を使用し、**+**、**-**、および ***** の 2 進演算を実行するという意味です。

注: NaN および無限大は **REAL(16)** に対して完全にはサポートされていません。算術演算は、これらの値を必ずしも伝搬するとは限りません。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: *X* が指定されていない場合、関数は false の値を戻します。

X が指定され、**REAL(16)** である場合には、関数は false の値を戻します。それ以外の場合は、関数は true を戻します。

例:

```
USE IEEE_ARITHMETIC
TYPE(IEEE_STATUS_TYPE) STATUS_VALUE
...
CALL IEEE_GET_STATUS(STATUS_VALUE)   ! Get status of all exception flags
CALL IEEE_SET_FLAG(IEEE_ALL,.FALSE.) ! Set all exception flags to quiet
... ! calculation involving exception handling
CALL IEEE_SET_STATUS(STATUS_VALUE)   ! Restore the flags
```

IEEE_SUPPORT_DENORMAL または IEEE_SUPPORT_DENORMAL(X)

照会 IEEE 関数。現行のインプリメンテーションが非正規値をサポートするかどうかを判別します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定されていない実数タイプのすべての引き数に対して、または *X* と同じ kind 型付きパラメーターの実変数に対して、インプリメンテーションが非正規値での算術演算および割り当てをサポートする場合には、結果は true の値を持ちます。それ以外の場合は、結果は false の値を持ちます。

IEEE_SUPPORT_DIVIDE または IEEE_SUPPORT_DIVIDE(X)

照会 IEEE 関数。現行のインプリメンテーションが IEEE 標準の精度の除算をサポートするかどうかを判別します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定されていない実数タイプのすべての引き数に対して、または *X* と同じ kind 型付きパラメーターの実変数に対して、インプリメンテーションが、IEEE 標準によって指定された精度での除算をサポートする場合には、結果は true の値を持ちます。それ以外の場合は、結果は false の値を持ちます。

IEEE_SUPPORT_FLAG(FLAG) または IEEE_SUPPORT_FLAG(FLAG, X)

照会 IEEE 関数。現行のインプリメンテーションが例外をサポートするかどうかを判別します。

モジュール: **IEEE_EXCEPTIONS**

構文: ここで、*FLAG* は **IEEE_FLAG_TYPE** のスカラー引き数です。*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: *X* が指定されていない実数タイプのすべての引き数に対して、または *X* と同じ kind 型付きパラメーターの実変数に対して、インプリメンテーションが、指定された例外の検出をサポートする場合には、結果は true の値を持ちます。それ以外の場合は、結果は false の値を持ちます。

X が指定されていない場合、結果は false の値を持ちます。

X が指定され、**REAL(16)** タイプである場合、結果は false の値を持ちます。それ以外の場合、結果は true の値を持ちます。

IEEE_SUPPORT_HALTING(FLAG)

照会 IEEE 関数。例外の発生後に実行を打ち切るための機能または継続するための機能を現行のインプリメンテーションがサポートするかどうかを判別します。現行のインプリメンテーションによるサポートには、**IEEE_SET_HALTING(FLAG)** を使用して停止モードを変更する機能が含まれます。

モジュール: **IEEE_EXCEPTIONS**

構文: ここで、*FLAG* は、**IEEE_FLAG_TYPE** の **INTENT(IN)** 引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: 結果はすべてのフラグに true の値を戻します。

IEEE_SUPPORT_INF または IEEE_SUPPORT_INF(X)

照会 IEEE 関数。現行のインプリメンテーションが IEEE 無限大機能をサポートするかどうかを判別します。サポートするとは、単項演算および 2 項演算 (組み込み関数および組み込みモジュール内の関数によって定義されるものを含む) の IEEE 無限大動作が IEEE 標準に準拠することを言います。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定されていない実数タイプのすべての引き数に対して、または *X* と同じ kind 型付きパラメーターの実変数に対して、インプリメンテーションが、IEEE の正と負の無限大をサポートする場合には、結果は true の値を持ちます。それ以外の場合は、結果は false の値を持ちます。

X が **REAL(16)** タイプである場合には、結果は false の値を持ちます。それ以外の場合、結果は true の値を持ちます。

IEEE_SUPPORT_IO または IEEE_SUPPORT_IO(X)

照会 IEEE 関数。現行のインプリメンテーションが IEEE ベースの変換丸めをサポートするかどうかを判別します。サポートするとは、*X* が不在の、実数タイプのすべての引

き数の **IEEE_UP**、**IEEE_DOWN**、**IEEE_ZERO** および **IEEE_NEAREST** モードに対して、または *X* と同じ kind 型付きパラメーターの実変数に対して、 IEEE 標準に記述された定様式 I/O で IEEE ベースの変換を行うための機能を言います。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定され、**REAL(16)** タイプである場合、結果は false の値を持ちます。それ以外の場合は、結果は true の値を戻します。

IEEE_SUPPORT_NAN または IEEE_SUPPORT_NAN(X)

照会 IEEE 関数。現行のインプリメンテーションが IEEE 非数値機能をサポートするかどうかを判別します。サポートするとは、単項演算および 2 進演算 (組み込み関数および組み込みモジュール内の関数によって定義されるものを含む) の IEEE NaN 動作が IEEE 標準に準拠するという意味です。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定されていない場合、結果は false の値を持ちます。

X が指定され、**REAL(16)** タイプである場合、結果は false の値を持ちます。それ以外の場合は、結果は true の値を戻します。

IEEE_SUPPORT_ROUNDING (ROUND_VALUE) または IEEE_SUPPORT_ROUNDING (ROUND_VALUE, X)

照会 IEEE 関数。現行のインプリメンテーションが実数タイプの引き数の特定の丸めモードをサポートするかどうかを判別します。サポートするとは、**IEEE_SET_ROUNDING_MODE** を使用した、丸めモードを変更する機能を言います。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*ROUND_VALUE* は **IEEE_ROUND_TYPE** のスカラー引き数です。*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定されていない場合、実数タイプのすべての引き数に対して、**ROUND_VALUE** によって定義された丸めモードをインプリメンテーションがサポートするときには、結果は true の値を持ちます。それ以外の場合は、結果は false の値を持ちます。

X が指定されている場合、 *X* と同じ kind 型付きパラメーターの実変数に対して、**ROUND_VALUE** によって定義された丸めモードをインプリメンテーションがサポートするときには、結果は true の値を戻します。

X が指定され、**REAL(16)** タイプである場合には、 **ROUND_VALUE** が **IEEE_NEAREST** の値を持つときには、結果は false の値を戻します。それ以外の場合は、結果は true の値を戻します。

ROUND_VALUE が **IEEE_OTHER** の値を持つときには、結果は false の値を持ちます。

IEEE_SUPPORT_SQRT または IEEE_SUPPORT_SQRT(X)

照会 IEEE 関数。現行のインプリメンテーションが IEEE 標準によって定義された **SQRT** をサポートするかどうかを判別します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

X が指定されていない場合、**REAL** タイプのすべての変数に関して IEEE 規則に **SQRT** が準拠するときには、結果は true の値を戻します。それ以外の場合は、結果は false の値を持ちます。

X が指定されている場合、*X* と同じ kind 型付きパラメーター付きの **REAL** タイプのすべての変数に関して、IEEE 規則に **SQRT** が準拠するときには、結果は true の値を戻します。

X が指定され、**REAL(16)** タイプである場合、結果は false の値を持ちます。それ以外の場合は、結果は true の値を戻します。

IEEE_SUPPORT_STANDARD または IEEE_SUPPORT_STANDARD(X)

照会 IEEE 関数。Fortran 2000 ドラフト標準で定義されたすべての機能がサポートされるかどうかを判別します。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は、実数タイプのスカラーまたは配列値引き数です。

結果タイプおよび属性: 結果は、デフォルト論理タイプのスカラーです。

規則: *X* が指定されていない場合、XL Fortran は **REAL(16)** をサポートするので、結果は false の値を戻します。

X が指定されている場合、以下の関数も true を戻せば、結果は true の値を戻します。
- **IEEE_SUPPORT_DATATYPE(X)**
- **IEEE_SUPPORT_DENORMAL(X)**
- **IEEE_SUPPORT_DIVIDE(X)**
- すべての有効フラグに対する **IEEE_SUPPORT_FLAG(FLAG, X)**。
- すべての有効フラグに対する **IEEE_SUPPORT_HALTING(FLAG)**。
- **IEEE_SUPPORT_INF(X)**
- **IEEE_SUPPORT_NAN(X)**
- すべての有効 **ROUND_VALUE** に対する **IEEE_SUPPORT_ROUNDING(ROUND_VALUE, X)**
- **IEEE_SUPPORT_SQRT(X)**

それ以外の場合は、結果は false の値を持ちます。

IEEE_UNORDERED(X, Y)

エレメント型 IEEE 非順序関数。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* および *Y* は実数タイプです。

結果タイプおよび属性: 結果のタイプは、デフォルトの論理値になります。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** および **IEEE_SUPPORT_DATATYPE(Y)** は、 true の値を戻す必要があります。

X または *Y* が NaN である場合に、 true の値を戻す非順序関数。それ以外の場合は、関数は false の値を戻します。

例:

```
USE IEEE_ARITHMETIC
REAL X, Y
X = 0.0
Y = IEEE_VALUE(Y, IEEE_QUIET_NAN)
PRINT *, IEEE_UNORDERED(X,Y)   ! Prints true
END
```

IEEE_VALUE(X, CLASS)

エレメント型 IEEE 関数。*CLASS* によって指定された IEEE 値を生成します。

注: 各種のプラットフォームで NaN 処理が異なるので、この関数のインプリメンテーションはプラットフォームおよびコンパイラーに依存します。バイナリー・ファイルに保管された NaN 値を、値を生成したプラットフォームと異なるプラットフォームで読み取ると、結果は未指定になります。

モジュール: **IEEE_ARITHMETIC**

構文: ここで、*X* は実数タイプです。 *CLASS* は **IEEE_CLASS_TYPE** タイプです。

結果タイプおよび属性: 結果は *X* と同じタイプおよび kind です。

規則: Fortran 2000 ドラフト標準に確実に準拠するようにするには、**IEEE_SUPPORT_DATATYPE(X)** は、true の値を戻す必要があります。

CLASS の値が **IEEE_SIGNALING_NAN** または **IEEE_QUIET_NAN** である場合には、**IEEE_SUPPORT_NAN(X)** は true でなくてはいけません。

CLASS の値が **IEEE_NEGATIVE_INF** または **IEEE_POSITIVE_INF** である場合には、**IEEE_SUPPORT_INF(X)** は true でなくてはいけません。

CLASS の値が **IEEE_NEGATIVE_DENORMAL** または **IEEE_POSITIVE_DENORMAL** である場合には、 **IEEE_SUPPORT_DENORMAL(X)** は true でなくてはいけません。

IEEE_VALUE(X, CLASS) を複数回呼び出すとき、 kind 型付きパラメーターおよび *CLASS* が同じままである場合には、特定の *X* 値に対して同じ結果を戻します。

コンパイル単位が **IEEE_SIGNALING_NAN** の *CLASS* 値でこのプログラムを呼び出す場合、コンパイル単位は **-qfloat=nans** コンパイラー・オプションでコンパイルする必要があります。

例:

```
USE IEEE_ARITHMETIC
REAL :: X
IF (IEEE_SUPPORT_DATATYPE(X)) THEN
   X = IEEE_VALUE(X, IEEE_NEGATIVE_INF)
   PRINT *, X ! Prints -inf
END IF
```

浮動小数点状況に関する規則

シグナル通知に設定された例外フラグは、**IEEE_SET_FLAG** または **IEEE_SET_STATUS** サブルーチンで静止に設定するまで、シグナル通知のままです。

コンパイラーは、**IEEE_EXCEPTIONS** または **IEEE_ARITHMETIC** 組み込みモジュールを使用する有効範囲単位からの呼び出しで、シグナル通知に例外フラグを設定する以外の方法では、浮動小数点数状況が変更されないようにしています。

IEEE_EXCEPTIONS または **IEEE_ARITHMETIC** モジュールを使用する有効範囲単位への入り口でフラグがシグナル通知に設定されると、そのフラグは静止に設定された後、その有効範囲単位を出るときにシグナル通知に復元されます。

IEEE_EXCEPTIONS または **IEEE_ARITHMETIC** モジュールを使用する有効範囲単位では、入り口で丸めモードおよび停止モードが変更されることはありません。戻るときには、丸めモードおよび停止モードは入り口のときと同じになります。

宣言式を評価で、例外でシグナル通知が発生する可能性があります。

例外ハンドラーは、**IEEE_EXCEPTIONS** または **IEEE_ARITHMETIC** モジュールを使用してはいけません。

以下の規則は、形式処理および組み込みプロシージャーに適用されます。

- シグナル通知フラグの状況は、シグナル通知であっても、または静止であっても、結果に反映されない中間計算によって変更されることはありません。
- 組み込みプロシージャーが正常に実行された場合には、フラグ **IEEE_OVERFLOW**、**IEEE_DIVIDE_BY_ZERO**、および **IEEE_INVALID** の値は、プロシージャーの入り口では同じままです。
- 実数または複素数の結果が組み込みには大きすぎて処理できない場合には、**IEEE_OVERFLOW** をシグナル通知することがあります。
- 実数または複素数の結果が、無効演算が原因で NaN である場合には、**IEEE_INVALID** をシグナル通知することがあります。

IEEE_GET_FLAG、**IEEE_SET_FLAG**、**IEEE_GET_STATUS**、**IEEE_SET_HALTING**、または **IEEE_SET_STATUS** の呼び出しがない一連のステートメントでは、以下が適用されます。演算の実行によってシグナル通知する例外が発生したが、そのシーケンスを実行した後、変数の値が演算に依存しない場合には、例外がシグナル通知されるかどうかは、最適化レベルに依存します。最適化の変換によって一部のコードが除去される場

合があります。そのため、除去されたコードによってシグナル通知された IEEE 例外フラグはシグナル通知されないことになります。

例外は、標準仕様で要求されるか許可される演算の範囲を超える演算の実行の間にだけ、その例外が発生しうる場合には、シグナル通知しません。

Fortran 以外によって定義されたプロシージャーでは、浮動小数点状況を保持するのはユーザーの責任です。

拡張精度値の場合、XL Fortran で常に浮動小数点演算の例外条件が検出されるわけではありません。また、拡張精度を使用するプログラムで浮動小数点演算例外のトラップをオンにすると、例外が実際には発生していない場合でも、シグナルが生成されることがあります。詳細については、「ユーザーズ・ガイド」の『*浮動小数点演算例外の検出とトラップ*』を参照してください。

Fortran 2000 IEEE 派生型、定数、および演算子は、**xlf_fp_util**、**fpsets**、および **fpgets** プロシージャーの浮動小数点および照会プロシージャーと互換性がありません。IEEE プロシージャーから取得した値は、非 IEEE プロシージャーでは使用できません。1 つの有効範囲単位内で、**xlf_fp_util**、**fpsets**、および **fpgets** のプロシージャー呼び出しと IEEE プロシージャー呼び出しを混合しないでください。これらのプロシージャーは、 **IEEE_EXCEPTIONS** または **IEEE_ARITHMETIC** モジュールを使用する有効範囲単位から呼び出されるときに、浮動小数点状況を変更します。

例

例 1: 次の例で、メインプログラムは、**IEEE_ARITHMETIC** モジュールを使用するプロシージャー *P* を呼び出します。このプロシージャーは、戻る前に、浮動小数点状況を変更します。例では、プロシージャー *P* を呼び出す前、プロシージャーへの入り口、*P* からの出口、およびプロシージャーからの戻り後における、浮動小数点状況の変更が示されています。

```
PROGRAM MAIN
  USE IEEE_ARITHMETIC

  INTERFACE
    SUBROUTINE P()
      USE IEEE_ARITHMETIC
    END SUBROUTINE P
  END INTERFACE

  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

  CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL P()
```

```
  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  IF (ROUND_VALUE == IEEE_NEAREST) THEN
    PRINT *, "MAIN: ROUNDING MODE: IEEE_NEAREST"
  ENDIF
END PROGRAM MAIN

SUBROUTINE P()
  USE IEEE_ARITHMETIC
  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "   P: FLAGS ON ENTRY: ",FLAG_VALUES

  CALL IEEE_SET_ROUNDING_MODE(IEEE_TO_ZERO)
  CALL IEEE_SET_FLAG(IEEE_UNDERFLOW, .TRUE.)

  CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  IF (ROUND_VALUE == IEEE_TO_ZERO) THEN
    PRINT *, "   P: ROUNDING MODE ON EXIT: IEEE_TO_ZERO"
  ENDIF
  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "   P: FLAGS ON EXIT: ",FLAG_VALUES
END SUBROUTINE P
```

IEEE 演算の規則に確実に準拠するようにするために **-qstrictieeemod** コンパイラー・オプションを使用するとき、 *P* を呼び出す前に設定された例外フラグは *P* への入り口でクリアされます。*P* で発生する浮動小数点状況の変更は、*P* が戻るときに取り消されます。ただし、例外として、*P* で設定されたフラグは、*P* が戻った後も設定されたままです。

```
 MAIN: FLAGS  T F F F F
    P: FLAGS ON ENTRY:  F F F F F
    P: ROUNDING MODE ON EXIT: IEEE_TO_ZERO
    P: FLAGS ON EXIT:  F F F T F
 MAIN: FLAGS  T F F T F
 MAIN: ROUNDING MODE: IEEE_NEAREST
```

-qnostrictieeemod コンパイラー・オプションが有効である場合、 *P* を呼び出す前に設定した例外フラグは、*P* への入り口でも設定されたままです。*P* で発生する浮動小数点状況の変更は呼び出し側に伝搬されます。

```
 MAIN: FLAGS  T F F F F
    P: FLAGS ON ENTRY:  T F F F F
    P: ROUNDING MODE ON EXIT: IEEE_TO_ZERO
    P: FLAGS ON EXIT:  T F F T F
 MAIN: FLAGS  T F F T F
```

例 2: 次の例で、メインプログラムは、**IEEE_ARITHMETIC** と **IEEE_EXCEPTIONS** のいずれも使用しないプロシージャー *Q* を呼び出します。プロシージャー *Q* は、戻る前に、浮動小数点状況を変更します。例では、*Q* を呼び出す前、プロシージャーへの入り口、Q からの出口、およびプロシージャーからの戻り後における浮動小数点状況の変更が示されています。

```
PROGRAM MAIN
  USE IEEE_ARITHMETIC

  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  TYPE(IEEE_ROUND_TYPE) :: ROUND_VALUE

  CALL IEEE_SET_FLAG(IEEE_OVERFLOW, .TRUE.)

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL Q()

  CALL IEEE_GET_FLAG(IEEE_ALL, FLAG_VALUES)
  PRINT *, "MAIN: FLAGS ",FLAG_VALUES

  CALL IEEE_GET_ROUNDING_MODE(ROUND_VALUE)
  IF (ROUND_VALUE == IEEE_NEAREST) THEN
    PRINT *, "MAIN: ROUNDING MODE: IEEE_NEAREST"
  ENDIF
END PROGRAM MAIN

SUBROUTINE Q()
  USE XLF_FP_UTIL
  INTERFACE
    FUNCTION GET_FLAGS()
      LOGICAL, DIMENSION(5) :: GET_FLAGS
    END FUNCTION
  END INTERFACE

  LOGICAL, DIMENSION(5) :: FLAG_VALUES
  INTEGER(FP_MODE_KIND) :: OLDMODE

  FLAG_VALUES = GET_FLAGS()
  PRINT *, "   Q: FLAGS ON ENTRY: ", FLAG_VALUES

  CALL CLR_FPSCR_FLAGS(FP_OVERFLOW)
  OLDMODE = SET_ROUND_MODE(FP_RND_RZ)
  CALL SET_FPSCR_FLAGS(TRP_OVERFLOW)
  CALL SET_FPSCR_FLAGS(FP_UNDERFLOW)

  IF (GET_ROUND_MODE() == FP_RND_RZ) THEN
    PRINT *, "   Q: ROUNDING MODE ON EXIT: TO_ZERO"
  ENDIF

  FLAG_VALUES = GET_FLAGS()
  PRINT *, "   Q: FLAGS ON EXIT: ", FLAG_VALUES
END SUBROUTINE Q
```

```
! PRINT THE STATUS OF ALL EXCEPTION FLAGS
FUNCTION GET_FLAGS()
  USE XLF_FP_UTIL
  LOGICAL, DIMENSION(5) :: GET_FLAGS
  INTEGER(FPSCR_KIND), DIMENSION(5) :: FLAGS
  INTEGER I

  FLAGS = (/ FP_OVERFLOW, FP_DIV_BY_ZERO, FP_INVALID, &
  &          FP_UNDERFLOW, FP_INEXACT /)
  DO I=1,5
    GET_FLAGS(I) = (GET_FPSCR_FLAGS(FLAGS(I)) /= 0)
  END DO
END FUNCTION
```

IEEE 演算の規則に確実に準拠するようにするために **-qstrictieeemod** コンパイラー・オプションを使用するとき、 *Q* の前に設定された例外フラグは *Q* への入り口で設定されたままです。 *Q* で発生する、浮動小数点状況の変更は *Q* が戻るときに取り消されます。ただし、例外として、*Q* で設定されたフラグは、*Q* が戻った後も設定されたままです。

```
MAIN: FLAGS  T F F F F
    Q: FLAGS ON ENTRY:  T F F F F
    Q: ROUNDING MODE ON EXIT: TO_ZERO
    Q: FLAGS ON EXIT:  F F F T F
 MAIN: FLAGS  T F F T F
 MAIN: ROUNDING MODE: IEEE_NEAREST
```

-qnostrictieeemod オプションが有効である場合、 *Q* を呼び出す前に設定した例外フラグは、*Q* への入り口でも設定されたままです。*Q* で発生する浮動小数点状況の変更は呼び出し側に伝搬されます。

```
MAIN: FLAGS  T F F F F
    Q: FLAGS ON ENTRY:  T F F F F
    Q: ROUNDING MODE ON EXIT: TO_ZERO
    Q: FLAGS ON EXIT:  F F F T F
 MAIN: FLAGS  F F F T F
```

IBM 拡張 の終り

第 17 章 サービス・プロシージャーおよびユーティリティー・プロシージャー

IBM 拡張

XL Fortran は、 Fortran プログラマーが使用できるユーティリティー・サービスを提供します。この節では、一般的なサービス・プロシージャーおよびユーティリティーについて説明し、次に、これらのプロシージャーのアルファベット順の参照を記載します。

一般的なサービス・プロシージャーおよびユーティリティー・プロシージャー

浮動小数点制御および照会のための効果的なプロシージャーは、`xlf_fp_util` モジュールに属します。一般的なサービス・プロシージャーおよびユーティリティー・プロシージャーは、`xlfutility` モジュールに属します。関数を正しいタイプで指定し、名前の競合を避けるために、これらのプロシージャーを使用する場合は、次の 2 つの方法のいずれかに従ってください。

1. XL Fortran は `XLFUTILITY` モジュールを提供します。このモジュールには、これらのプロシージャー用のインターフェースおよびデータ型定義 (および **dtime_**、**etime_**、**idate_**、**itime_** の各プロシージャーに必要な派生型定義) が含まれています。 XL Fortran は、タイプ、種類 (kind)、およびランクがインターフェース仕様と互換性のない引き数にフラグを付けます。これらのモジュールを使用することにより、リンク時まで待たずに、コンパイル時にこれらのプロシージャーのタイプ・チェックを行うことができます。モジュール・インターフェース内の引き数名は、以下に定義する例からとられています。 `xlfutility` および `xlfutility_extname` の 2 つのモジュールに対して、次のファイルが提供されています。

ファイル名	ファイル・タイプ	場所
• xlfutility.f • xlfutility_extname.f	ソース・ファイル	• /usr/lpp/xlf/samples/modules
• xlfutility.mod • xlfutility_extname.mod	モジュール・シンボル・ファイル (32 ビット)	• /usr/lpp/xlf/include_32_d10 • /usr/lpp/xlf/include_32_d7 **注:** これらのディレクトリー内のファイルは、それぞれまったく同じものです。
	モジュール・シンボル・ファイル (64 ビット)	• /usr/lpp/xlf/include64

ソース・ファイルに **USE** ステートメントを追加することによって、プリコンパイルされたモジュールを使用することができます (詳細については、 488 ページの『USE』を参照してください)。また、必要に応じてモジュール・ソース・ファイルを修正し、再コンパイルすることもできます。 **-qextname** オプションを使用してコンパイルするプロシージャーの場合は、`xlfutility_extname` ファイルを使用してください。ソース・ファイル `xlfutility_extname.f` の場合、プロシージャー名の後に下線は付きませんが、`xlfutility.f` の場合は一部のプロシージャー名 (この章に記載されています) に下線が含まれています。

名前の競合が生じる場合 (たとえば、アクセス元のサブプログラムにモジュール・エンティティーと同じ名前のエンティティーがある場合) は、**ONLY** 文節を使用するか、または **USE** ステートメントの名前変更機能を使用してください。以下に例を示します。

```
USE XLFUTILITY, NULL1 => DTIME_, NULL2 => ETIME_
```

2. これらのプロシージャーは組み込みプロシージャーではないので、ユーザーは次のことを行ってください。
   - 暗黙のタイプ指定の潜在的な問題を回避するために、タイプを宣言します。
   - **-U** オプションを指定してコンパイルする場合は、これらのプロシージャーの名前をすべて小文字でコーディングして、XL Fortran ライブラリー内の名前と一致させる必要があります。このことを忘れないようにするために、ここでは小文字で名前を記します。

libc ライブラリー内の名前との競合を回避するため、一部のプロシージャー名には最後に下線が付いています。これらのプロシージャーへの呼び出しをコーディングする場合は、以下のことができます。

- 下線を入力する代わりに、**-brename** リンカー・オプションを使用して、リンク時に名前を変更します。

  ```
  xlf -brename:flush,flush_
  calls_flush.f
  ```

  この方法が最もよく機能するのは、少数のプロシージャーを名前変更するだけでよい場合です。
- 下線を入力する代わりに、**-qextname** コンパイラー・オプションを使用して、個々の名前の終わりに下線を追加します。

  ```
  xlf -qextname calls_flush.f
  ```

  この方法は、ルーチン名の後に下線を付けずにすでに書かれているプログラムに対して使用することをお勧めします。 XL Fortran ライブラリーには、**fpgets_** などのエントリー・ポイントがさらに含まれているので、後続下線を使用しないプロシージャーへの呼び出しは、依然として **-qextname** で解決します。

- プログラムの構成方法、および、プログラムが使用している特定のライブラリー・ファイルおよびオブジェクト・ファイルによっては、 **-qextname** または **-brename** の使用が困難な場合があります。このような場合には、ソース・ファイル内の該当する名前の後に下線を入力してください。

```
PRINT *, IRTC()  ! No underscore in this name
CALL FLUSH_(10)  ! But there is one in this name
```

お使いのプログラムが以下のプロシージャーを呼び出す場合は、使用できる共通ブロックおよび外部プロシージャー名に制限があります。

XLF 提供の関数名	使用できない共通ブロックまたは外部プロシージャー名
mclock	times
rand	irand

注: XL Fortran バージョン 2 における **mvbits** サブルーチンは、現在は、組み込みサブルーチン (637 ページの『MVBITS (FROM, FROMPOS, LEN, TO, TOPOS)』) になっています。

サービス・プロシージャーおよびユーティリティー・プロシージャーのリスト

この節には、XLFUTILITY モジュールのサービス・プロシージャーとユーティリティー・プロシージャーがリストされています。

プロシージャー **ctime**、**gmtime_**、**ltime_**、または **time_** のインターフェースを使用する 64 ビット・アプリケーションは、特定の組み込みデータ型の kind 型付きパラメーターを指定するためにシンボリック定数 TIME_SIZE を使用します。 TIME_SIZE は、XLFUTILITY モジュール・ファイルに定義されています。これは、32 ビット・アプリケーションと 64 ビット非 LDT アプリケーションでは「4」に、または 64 ビット LDT アプリケーションでは「8」に設定されます。

注: CHARACTER(n) は、その変数に対して任意の長さを指定できることを意味します。

alarm_

alarm_ 関数は、時刻 TIME にアラーム・シグナルを送信して、関数 SUB を呼び出します。

引き数タイプおよび属性

INTEGER(4)

結果タイプおよび属性

INTEGER(4)

結果値

戻される値は、最後のアラームから残っている時間です。

例

```
INTEGER(4) REMAINING, TIME, alarm_
INTERFACE
  SUBROUTINE SUB
  END SUBROUTINE SUB
END INTERFACE
REMAINING =  alarm_ (TIME, SUB)
```

bic_

bic_ サブルーチンは、ビット *X2* の *X1* を 0 に設定します。

引き数タイプおよび属性

INTEGER(4)

X1 は、範囲 $0 \leq X1 \leq 31$ の値を持ちます。

例

```
INTEGER(4) X1, X2
CALL bic_ (X1, X2)
```

bis_

bis_ サブルーチンは、ビット *X2* の *X1* を 1 に設定します。

引き数タイプおよび属性

INTEGER(4)

X1 は、範囲 $0 \leq X1 \leq 31$ の値を持ちます。

例

```
INTEGER(4) X1, X2
CALL bis_ (X1, X2)
```

bit_

bit_ 関数は、ビット *X2* の *X1* が 1 に等しければ、値 **.TRUE.** を戻します。そうでない場合は、**bit_** は値 0 を戻します。

引き数タイプおよび属性

INTEGER(4)

X1 は、範囲 0 ≤ *X1* ≤ 31 の値を持ちます。

結果タイプおよび属性

LOGICAL(4)

例

```
INTEGER(4) X2, X1
LOGICAL BITK, bit_
BITK = bit_ (X1, X2)
```

clock_

clock_ 関数は、hh:mm:ss 形式で時刻を戻します。この関数は、オペレーティング・システムのクロック関数とは異なります。

結果タイプおよび属性

長さが 8 の文字

結果値

hh:mm:ss 形式の時刻

例

```
CHARACTER(8) C, clock_
C = clock_()
```

ctime_

ctime_ サブルーチンは、システム時刻 TIME を 26 文字の ASCII ストリングに変換し、結果を最初の引き数に出力します。

引き数タイプおよび属性

最初の引き数は、長さが 26 の文字です。 2 番目の引き数は、INTEGER(4) です。

例

```
INTEGER(KIND=TIME_SIZE) TIME
CHARACTER(26) STR
CALL ctime_(STR, TIME)
```

date

date 関数は mm/dd/yy 形式で現在の日付を戻します。

結果タイプおよび属性

長さが 8 の文字

結果値

mm/dd/yy 形式の現在日付

例

```
CHARACTER(8) D, date
D = date()
```

dtime_

dtime_ 関数は、 DTIME_STRUCT にユーザー時間とシステム時間の時間会計情報を設定します。時間測定はすべて、1/100 秒単位で行われます。出力の表示単位は秒です。

結果タイプおよび属性

長さが 4 の実数

結果値

戻される値は、**dtime_** の最後の呼び出し以降のユーザー時間とシステム時間の合計です。

例

```
REAL(4) DELTA, dtime_
TYPE TB_TYPE
  SEQUENCE
  REAL(4) USRTIME
  REAL(4) SYSTIME
END TYPE
TYPE (TB_TYPE) DTIME_STRUCT
DELTA = dtime_(DTIME_STRUCT)
```

etime_

etime_ 関数は、プロセスの実行開始以降のユーザー経過時間とシステム経過時間を ETIME_STRUCT に設定します。時間測定はすべて、1/100 秒単位で行われます。出力の表示単位は秒です。

結果タイプおよび属性

長さが 4 の実数

結果値

戻される値は、ユーザー経過時間とシステム経過時間の合計です。

例

```
REAL(4) ELAPSED, etime_
TYPE TB_TYPE
  SEQUENCE
  REAL(4) USRTIME
  REAL(4) SYSTIME
END TYPE
TYPE (TB_TYPE) ETIME_STRUCT
ELAPSED = etime_(ETIME_STRUCT)
```

exit_

exit_ サブルーチンは、プロセスの実行を停止して、 EXIT_STATUS の終了状況を通知します。

引き数タイプおよび属性

INTEGER(4)

例

```
INTEGER(4) EXIT_STATUS
CALL exit_(EXIT_STATUS)
```

fdate_

fdate_ サブルーチンは、 26 文字の ASCII 文字ストリングで日付と時刻を戻します。この例では、日付と時刻は STR で戻されます。

引き数タイプおよび属性

引き数は、長さが 26 の文字です。

例

```
CHARACTER(26) STR
CALL fdate_(STR)
```

fiosetup_

fiosetup_ 関数は、UNIT で指定された論理装置に対する、要求された I/O 動作の設定を行います。要求は、引き数 COMMAND で指定します。引き数 ARGUMENT は、COMMAND への引き数です。 Fortran インクルード・ファイル fiosetup_.h がコンパイラーとともに提供されており、fiosetup_ の引き数とエラー戻りコードに対するシンボリック定数を定義しています。

UNIT
現在ファイルに接続されている論理装置です。

COMMAND
IO_CMD_FLUSH_AFTER_WRITE (1)。 すべての WRITE ステートメントの後で、指定した UNIT のバッファーがフラッシュされるかどうかを指定します。

IO_CMD_FLUSH_BEFORE_READ (2)。 すべての READ ステートメントの前に、指定した UNIT のバッファーがフラッシュされるかどうかを指定します。 これは、現在バッファー内にあるデータをリフレッシュするのに使用することができます。

ARGUMENT
IO_ARG_FLUSH_YES (1)。 すべての WRITE ステートメントの後で、指定された UNIT のバッファーをフラッシュします。この引き数は、コマンド IO_CMD_FLUSH_AFTER_WRITE および IO_CMD_FLUSH_BEFORE_READ とともに指定します。

IO_ARG_FLUSH_NO (0)。I/O ライブラリーに、独自の判断でバッファーをフラッシュするよう指示します。 ある特定の装置タイプに接続された装置は、 IO_CMD_FLUSH_AFTER_WRITE の設定値にかかわりなく、各 WRITE 操作の後にフラッシュしなければならないことに注意してください。そのような装置には、端末およびパイプがあります。この引き数は、コマンド IO_CMD_FLUSH_AFTER_WRITE および IO_CMD_FLUSH_BEFORE_READ とともに指定します。これが、両方のコマンドにおけるデフォルト設定です。

結果タイプおよび属性

結果値

例

```
FUNCTION fiosetup_(UNIT, COMMAND, ARGUMENT)
INTEGER(4) fiosetup_, IRESULT
INTEGER(4) UNIT, COMMAND, ARGUMENT
INCLUDE 'fiosetup_.h'
OPEN ( UNIT=42, FILE="foo", ...)
IRESULT = fiosetup_(42, &
     IO_CMD_FLUSH_AFTER_WRITE, &
     IO_ARG_FLUSH_YES)
```

```
The service routine FIOSETUP_ returns 0 if it
succeeds. Otherwise, it returns
one of the following error codes:
```

```
IO_ERR_NO_RTE (1000) the run-time
environment is not running.
IO_ERR_BAD_UNIT(1001) 指定された UNIT が接続されていない。
IO_ERR_BAD_CMD (1002) 無効なコマンド。
IO_ERR_BAD_ARG (1003) 無効な引き数。
```

flush_

flush_ サブルーチンは、論理装置 LUNIT 用の I/O バッファーの内容をフラッシュします。 LUNIT の値は、0 ≤ LUNIT ≤ 2**31-1 の範囲内になければなりません。

引き数タイプおよび属性

INTEGER(4)

例

```
INTEGER(4) LUNIT
CALL flush_(LUNIT)
```

ftell_ ftell64_

ftell_ 関数は、指定した論理装置 UNIT と関連のあるファイルの先頭と比較しての現行バイトのオフセットを戻します。 装置が接続されていない場合、**ftell_** 関数は -1 を戻します。

ftell64_ 関数は、例外として、この関数が、ラージ・ファイル使用可能ファイル・システムにおいて、2 ギガバイトより大きなファイルでも機能すること以外は、**ftell_** 関数と同一です。

ftell_ または **ftell64_** 関数から戻されたオフセットは、前に完了した I/O オペレーションの結果です。 同一の装置でのすべての未解決の非同期データ転送操作において、対応する **WAIT** ステートメントが実行されるまで、未解決の非同期データ転送操作による、その装置での **ftell_** または **ftell64_** への参照はできません。

ftell_ または **ftell64_** 関数によって戻されるオフセットは、ファイルの先頭からの相対的な現行バイトの絶対オフセットです。 つまり、ファイルの先頭から現行バイトまでのすべてのバイトがカウントされます。データのレコードとレコード終了文字がある場合、それらも含まれます。

引き数タイプおよび属性

INTEGER(4)

結果タイプおよび属性

ftell_ は INTEGER(4) を戻します。

ftell64_ は INTEGER(8) を戻します。

例

```
INTEGER(4) ftell_, UNIT1, UNIT2, IRESULT
INTEGER(8) ftell64_, IRESULT8

UNIT1 = 42
IRESULT = ftell_(UNIT1)

UNIT2 = 44
IRESULT8 = ftell64_(UNIT2)
! Unit 44 might be connected to a
! file larger than 2 gigabytes
```

getarg

getarg サブルーチンは、現行プロセスのコマンド行引き数を戻します。 I1 は、どのコマンド行引き数を戻すかを指定する整数引き数です。 C1 は文字タイプの引き数で、**getarg** からの戻り時では、この引き数にはコマンド行引き数が含まれています。 I1 が 0 に等しい場合は、プログラム名が戻されます。

引き数タイプおよび属性

最初の引き数は、INTEGER(4) です。 2 番目の引き数は、文字ストリングです。

例

```
INTEGER(4) I1
CHARACTER(n) C1
CALL getarg(I1,C1)
```

getcwd_

getcwd_ 関数は、現行作業ディレクトリーのパス名 NAME を検索します。このパス名の最大長は 1024 文字です。

引き数タイプおよび属性

引き数は、最大長が 1024 の文字です。

結果タイプおよび属性

INTEGER(4)

結果値

戻される値は、正常終了した場合は 0 で、それ以外はエラー番号になります。

例

```
INTEGER(4) IS_CWD, getcwd_
CHARACTER(1024) NAME
IS_CWD = getcwd_ (NAME)
```

---

getfd

特定の Fortran 論理装置を指定すると、 **getfd** 関数はその装置の基本をなすファイル記述子を戻し、その装置が接続されていない場合には -1 を戻します。

注: XL Fortran は独自の I/O バッファリングを行うため、この関数を使用する際は特別な注意が必要となる場合があります。これについては、「ユーザーズ・ガイド」の『*混合言語の I/O*』で説明しています。

引き数タイプおよび属性

INTEGER(4)

結果タイプおよび属性

INTEGER(4)

結果値

指定された論理装置の基礎となるファイル記述子。

例

```
INTEGER(4) LUNIT, FD, getfd
FD = getfd(LUNIT)
```

---

getgid_

getgid_ 関数は、プロセスのグループ ID を戻します。この GROUP_ID は、呼び出しプロセスの要求された実グループ ID です。

結果タイプおよび属性

長さが 4 の整数

結果値

プロセスのグループ ID

例

```
INTEGER(4) GROUP_ID, getgid_
GROUP_ID = getgid_()
```

getlog_

getlog_ サブルーチンは、ユーザーのログイン名を NAME に格納します。 NAME の最大長は 8 文字です。ユーザーのログイン名が見つからない場合は、NAME はブランクで埋め込まれます。

引き数タイプおよび属性

引き数は、最大長が 8 の文字です。

例

```
CHARACTER(8) NAME
CALL getlog_ (NAME)
```

getpid_

getpid_ 関数は、現行プロセスのプロセス ID を PROCESS_ID に戻します。

結果タイプおよび属性

長さが 4 の整数

結果値

現行プロセスのプロセス ID

例

```
INTEGER(4) PROCESS_ID, getpid_
PROCESS_ID = getpid_()
```

getuid_

getuid_ 関数は、現行プロセスの実ユーザー ID を USER_ID に戻します。

結果タイプおよび属性

長さが 4 の整数

結果値

現行プロセスの実ユーザー ID

例

```
INTEGER(4) USER_ID, getuid_
USER_ID = getuid_()
```

global_timef

global_timef 関数は、すべての実行中のスレッドにおいて **global_timef** への最初の呼び出しが最初に行われた時点以降の経過時間をミリ秒単位で戻します。スレッド固有の時間計測結果については、『timef 関数』を参照してください。

結果タイプおよび属性

長さが 8 の実数

結果値

この関数は、すべての実行中のスレッドからのグローバル時間計測結果を戻します。**global_timef** への最初の呼び出しは、0.0 (ミリ秒単位) を戻します。

例

```
INTEGER N
REAL(8) global_timef, T1, T2, T3
  T1 = global_timef()   ! returns 0.0
    DO I = 1, N         ! loop 1
      H = I + 1000
    END DO
    DO I = 1, N         ! loop 2
      M = I + 2000
    END DO
  T2 = global_timef()
! returns the elapsed time of
!  loop 1 and loop 2
    DO I = 1, N         ! loop 3
      M = I + 3000
    END DO
  T3 = global_timef()
! returns the elapsed time of
!  loop 1, 2 and 3
END
```

gmtime_

gmtime_ サブルーチンは、システム時刻 STIME を配列 TARRAY に変換します。データは以下の順で TARRAY に格納されます。

秒 (0 ～ 59)
分 (0 ～ 59)
時間 (0 ～ 23)
月間通算日 (1 ～ 31)
月 (0 ～ 11)
年 (年 = 現在の年 - 1900)
曜日 (日曜日 = 0)
年間通算日 (0 ～ 365)
夏時間 (0 または 1)

引き数タイプおよび属性

最初の引き数は、INTEGER(4) です。 2 番目の引き数は、ランク 1 でサイズ 9 の INTEGER(4) 配列です。

例

```
INTEGER(KIND=TIME_SIZE) STIME
INTEGER(4) TARRAY(9)
CALL gmtime_(STIME, TARRAY)
```

hostnm_

hostnm_ 関数は、マシンのホスト名 NAME を検索します。 NAME の最大長は 32 文字です。

引き数タイプおよび属性

引き数は、最大長が 32 の文字です。

結果タイプおよび属性

INTEGER(4)。

結果値

戻される値は、ホスト名が見つかった場合は 0 で、それ以外の場合はエラー番号になります。

例

```
INTEGER(4) ISHOST, hostnm_
CHARACTER(32) NAME
ISHOST = hostnm_ (NAME)
```

iargc

iargc 関数は、実行時にコマンド行に入力されたプログラム名の後の引き数の数を表す整数を戻します。

結果タイプおよび属性

長さが 4 の整数

結果値

引き数の数

例

```
INTEGER(4) I1, iargc
I1 = iargc()
```

idate_

idate_ サブルーチンは、日、月、年が入った数値形式で現在の日付を IDATE_STRUCT に戻します。

例

```
TYPE IDATE_TYPE
  SEQUENCE
  INTEGER(4) IDAY
  INTEGER(4) IMONTH
  INTEGER(4) IYEAR
END TYPE
TYPE (IDATE_TYPE) IDATE_STRUCT
CALL idate_(IDATE_STRUCT)
```

ierrno_

ierrno_ 関数は、最後に検出されたシステム・エラーのエラー番号 SYSERROR を戻します。

結果タイプおよび属性

長さが 4 の整数

結果値

最後に検出されたシステム・エラーのエラー番号

例

```
INTEGER(4) SYSERROR, ierrno_
SYSERROR = ierrno_()
```

irand

irand 関数は、1 以上で、32768 以下の正の整数を生成します。 677 ページの『SRAND (SEED)』の組み込みサブルーチンは、乱数発生ルーチンのシード値を提供するのに使用されます。

結果タイプおよび属性

長さが 4 の整数

結果値

1 以上で 32768 以下の正の乱数

例

```
INTEGER(4) I1, irand
CALL SRAND(I1)
I1 = irand()
```

irtc

irtc 関数は、マシンのリアルタイム・クロックの初期値以降のナノ秒数の INTEGER(8) 値を戻します。

結果タイプおよび属性

長さが 8 の整数

結果値

マシンのリアルタイム・クロックの初期値以降のナノ秒数

例

```
INTEGER(8) A, B, irtc
A = irtc()
DO M = 1,20000
  N = N + M
END DO
```

```
B = irtc()
! How many nanoseconds elapsed?
PRINT *, B - A
END
```

itime_

itime_ サブルーチンは、秒、分、時間が入った数値形式で現在の時刻を ITIME_STRUCT に戻します。

例

```
TYPE IAR
  SEQUENCE
  INTEGER(4) IHR
  INTEGER(4) IMIN
  INTEGER(4) ISEC
END TYPE
TYPE (IAR) ITIME_STRUCT
CALL itime_(ITIME_STRUCT)
```

jdate

jdate 関数は、yyddd 形式で現在の年間通算日を戻します。

結果タイプおよび属性

長さが 8 の文字

結果値

yyddd 形式の現在の年間通算日

例

```
CHARACTER(8) D, jdate
D = jdate()
```

lenchr_

lenchr_ 関数は、文字ストリング STR の長さを LENGTH に格納します。

引き数タイプおよび属性

引き数は、文字タイプです。

結果タイプおよび属性

INTEGER(4)

結果値

文字ストリングの長さ

例

```
INTEGER(4) LENGTH, lenchr_
CHARACTER*(*) STR
LENGTH = lenchr_(STR)
```

lnblnk_

lnblnk_ 関数は、文字ストリング STR 内の最後の非ブランク文字の指標 INDEX を戻します。ストリングに非ブランク文字が入っていない場合は、INDEX は 0 に設定されます。

引き数タイプおよび属性

引き数は、文字タイプです。

結果タイプおよび属性

INTEGER(4)

結果値

ストリングの内の最後の非ブランク文字の指標、または非ブランク文字がない場合は 0

例

```
INTEGER(4) INDEX, lnblnk_
CHARACTER(n) STR
INDEX = lnblnk_(STR)
```

ltime_

ltime_ サブルーチンは、システム時刻 STIME (秒単位) を、 GMT を含む配列 TARRAY に切り分けます。この場合、切り分けられた時間はローカルの時間帯用に修正されます。データは以下の順で TARRAY に格納されます。

秒 (0 ～ 59)
分 (0 ～ 59)
時間 (0 ～ 23)
月間通算日 (1 ～ 31)
月 (0 ～ 11)
年 (年 = 現在の年 - 1900)

曜日 (日曜日 = 0)
年間通算日 (0 ～ 365)
夏時間 (0 または 1)

引き数タイプおよび属性

引き数 1 のタイプは INTEGER(4) です。引き数 2 のタイプは INTEGER(4) 配列、ランクは 1、サイズは 9 です。

例

```
INTEGER(KIND=TIME_SIZE) STIME
INTEGER(4) TARRAY(9)
CALL ltime_(STIME, TARRAY)
```

mclock

mclock 関数は、現行プロセスおよびその子プロセスの時間会計情報を戻します。

結果タイプおよび属性

長さが 4 の整数

結果値

戻される値は、現行プロセスのユーザー時間、すべての子プロセスのユーザー時間とシステム時間の合計です。 計測単位は 1/100 秒です。

例

```
INTEGER(4) I1, mclock
I1 = mclock()
```

qsort_

qsort_ サブルーチンは、 1 次元の配列 ARRAY に対して並列クィック・ソートを実行します。この ARRAY の長さ LEN は配列内のエレメント数であり、各エレメントの大きさは ISIZE です。また、このサブルーチンは、ユーザー定義のソート順序関数 COMPAR を実行して、配列のエレメントをソートします。 COMPAR 関数の必要事項については「AIX Technical Reference: Base Operating System and Extensions Volume 2」の 『qsort サブルーチン』の項で説明されています。

例

```
INTEGER(4) FUNCTION COMPAR_UP(C1, C2)
INTEGER(4) C1, C2
IF (C1.LT.C2) COMPAR_UP = -1
IF (C1.EQ.C2) COMPAR_UP =  0
```

```
IF (C1.GT.C2) COMPAR_UP =  1
RETURN
END
SUBROUTINE FOO()
INTEGER(4) COMPAR_UP
EXTERNAL COMPAR_UP
INTEGER(4) ARRAY(8), LEN, ISIZE
DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
LEN = 6
ISIZE = 4
CALL qsort_(ARRAY(3:8), LEN, ISIZE, COMPAR_UP)
! sorting ARRAY(3:8)
PRINT *, ARRAY
! result value is [0, 3, 1, 2, 4, 5, 7, 9]
RETURN
END
```

qsort_down

qsort_down サブルーチンは、 1 次元の配列 ARRAY に対して並列クイック・ソートを実行します。この ARRAY の長さ LEN は配列内のエレメント数であり、各エレメントの大きさは ISIZE です。結果は、配列 ARRAY 内に降順で格納されます。 **qsort_** とは異なり、 **qsort_down** サブルーチンは COMPAR 関数を必要としません。

例

```
SUBROUTINE FOO()
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 8
  ISIZE = 4
  CALL qsort_down(ARRAY, LEN, ISIZE)
  PRINT *, ARRAY
! Result value is [9, 7, 5, 4, 3, 2, 1, 0]
RETURN
END
```

qsort_up

qsort_up サブルーチンは、連続した 1 次元の配列 ARRAY に対して並列クイック・ソートを実行します。この ARRAY の長さ LEN は配列内のエレメント数であり、各エレメントの大きさは ISIZE です。結果は、配列 ARRAY 内に昇順で格納されます。 **qsort_** とは異なり、 **qsort_up** サブルーチンは COMPAR 関数を必要としません。

例

```
SUBROUTINE FOO()
  INTEGER(4) ARRAY(8), LEN, ISIZE
  DATA ARRAY/0, 3, 1, 2, 9, 5, 7, 4/
  LEN = 8
  ISIZE = 4
```

```
  CALL qsort_up(ARRAY, LEN, ISIZE)
  PRINT *, ARRAY
! Result value is [0, 1, 2, 3, 4, 5, 7, 9]
RETURN
END
```

rtc

rtc 関数は、マシンのリアルタイム・クロックの初期値以降の秒数の REAL(8) 値を戻します。

結果タイプおよび属性

長さが 8 の実数

結果値

マシンのリアルタイム・クロックの初期値以降の秒数

例

```
REAL(8) A, B, rtc
A = rtc()
DO M = 1,20000
  N = N + M
END DO
B = rtc()
! How many seconds elapsed?
PRINT *, B - A
END
```

setrteopts

setrteopts サブルーチンはプログラムの実行中に、 1 つまたは複数の実行時オプションの設定を変更します。実行時オプションに関する詳細は、「ユーザーズ・ガイド」の『*実行時オプションの設定*』を参照してください。

引き数タイプおよび属性

引き数は、文字タイプです。

例

```
CHARACTER(n) C1
CALL setrteopts (C1)
! For example,
! CALL setrteopts &
!   ('langlvl=90std:cnverr=no')
```

sleep_

sleep_ サブルーチンは、現行プロセスの実行を SEC 秒間中断します。

引き数タイプおよび属性

INTEGER(4)

例

```
INTEGER(4) SEC
CALL sleep_(SEC)
```

time_

time_ 関数は現在の時刻 (GMT) CURRTIME を秒単位で戻します。

結果タイプおよび属性

長さが 4 の整数

結果値

秒単位での現在時刻 (GMT)

例

```
INTEGER(KIND=TIME_SIZE) CURRTIME, time_
CURRTIME = time_()
```

timef

timef 関数は、最初に timef が呼び出されたときからの経過時間をミリ秒で戻します。

結果タイプおよび属性

長さが 8 の実数

結果値

最初に timef が呼び出されたときからの経過時間 (ミリ秒)。 **timef** の最初の呼び出しは 0.0d0 を戻します。

例

```
REAL(8) ELAPSED, timef
ELAPSED = timef()
DO M = 1,20000
  A = A ** 2
ENDDO
ELAPSED = TIMEF()
```

timef_delta

timef_delta 関数は、同じスレッド内で、引き数を 0.0 に設定して最後のインスタンス **timef_delta** を呼び出した時からの経過時間をミリ秒で戻します。正確な経過時間を得るには、時間を計測したいスレッドの領域を決定しなければなりません。この領域は **timef_delta(T0)** への呼び出しで開始される必要があります。ここで TO は初期化されます (TO=0.0) 。 **timef_delta** への次の呼び出しは、経過時間を得たい場合には、その最初の呼び出しの戻り値を入力引き数として使用しなければなりません。

結果タイプおよび属性

長さが 8 の実数

例

```
REAL(8) timef_delta, T0, T1, T2
  T0 = 0.0
    DO I = 1, N        ! Loop 1
      H = I + 1000
    END DO
  T1 = timef_delta(T0)
    DO I = 1, N        ! T1 gives the
      M = I + 2000     ! starting time
    END DO             ! of loop 2
  T2 = timef_delta(T1)
    DO I = 1, N        ! T2 gives the
      M = I + 3000     ! elapsed time
    END DO             ! of loop 2
```

umask_

umask_ 関数は、ファイル・モード作成マスクを CMASK に設定します。

引き数タイプおよび属性

INTEGER(4)

結果タイプおよび属性

INTEGER(4)

結果値

戻される値は、ファイル・モード作成マスクの前回の値です。

例

```
INTEGER(4) CMASK, LASTMASK, umask_
LASTMASK = umask_ (CMASK)
```

usleep_

usleep_ 関数は、現行プロセスの実行を MSEC マイクロ秒の間隔で中断します。

引き数タイプおよび属性

INTEGER(4)

結果タイプおよび属性

INTEGER(4)

結果値

戻される値は、関数が正常終了した場合は 0 で、それ以外の場合はエラー番号になります。

例

```
INTEGER(4) IS_SLEEP, MSEC, usleep_
IS_SLEEP = usleep_ (MSEC)
```

xl_ _trbk

xl_ _trbk サブルーチンは、呼び出し点から始まるトレースバックを提供します。**xl_ _trbk** はユーザーのコードから呼び出すことができます。ただしシグナル・ハンドラーからは呼び出せません。このサブルーチンには、パラメーターは必要ありません。

例

```
INTEGER res, n
IF (n .EQ. 1) THEN
   res=1
   CALL XL_ _TRBK()
ELSE
   res=n * FACTORIAL(n-1)
ENDIF
```

IBM 拡張 の終り

第 4 部 ハードウェアと XL Fortran

次の章には、ハードウェアおよび XL Fortran コンパイラーに特有の情報が含まれています。

- ハードウェア・ディレクティブと組み込みプロシージャー

以下の部では、XL Fortran 言語のその他の特徴について説明しています。

- XL Fortran 言語
- XL Fortran でのマルチスレッド・プログラミング
- XL Fortran 言語ユーティリティー

第 18 章 ハードウェア・ディレクティブと組み込みプロシージャー

IBM 拡張

この節では、ハードウェア固有のコンパイラー・ディレクティブと組み込み関数をアルファベット順に説明します。 ディレクティブおよび組み込み関数の中には、**-qarch** コンパイラー・オプションの中でアーキテクチャーを指定しなければならないものがあります。サブオプション構文は、この章で文書化されているディレクティブと組み込み関数の特定の要件によって異なります。 特定のアーキテクチャー用のプログラムのコンパイルについては、「ユーザーズ・ガイド」を参照してください。

ハードウェア固有のディレクティブ

この節では、ハードウェア固有のコンパイラー・ディレクティブをアルファベット順に説明します。特に記載のない場合、ディレクティブはどのハードウェアでも機能します。この節では、以下のディレクティブを説明します。

『CACHE_ZERO』	906 ページの『LIGHT_SYNC』
906 ページの『ISYNC』	906 ページの『PREFETCH』

CACHE_ZERO

CACHE_ZERO ディレクティブは、マシン・インストラクション dcbz (data cache block set to zero) を起動します。この命令は、指定された変数に対応するデータ・キャッシュ・ブロックをゼロに設定します。このディレクティブは、慎重に使用してください。

構文

```
▶▶──CACHE_ZERO──(──cv_var_list──)──────────────────────────────▶◀
```

cv_var ゼロに設定されるキャッシュ・ブロックに関連した変数。変数は、判別可能なストレージ・アドレスを持つデータ・オブジェクトでなければなりません。変数は、プロシージャー名、サブルーチン名、モジュール名、関数名、定数、ラベル、ゼロ・サイズのストリング、またはベクトル添え字を持つ配列にすることはできません。

例

以下の例では、0 にしたいキャッシュ・ブロックに配列 *ARRA* がすでにロードされていると想定します。次に、キャッシュ・ブロック内のデータがゼロに設定されます。

```
  real(4) :: arrA(2**5)
! ....
!IBM* CACHE_ZERO(arrA(1))            ! set data in cache block to zero
```

ISYNC

ISYNC ディレクティブを使用すると、すべての先行する命令が完了した後に、プリフェッチされた命令を破棄することができます。後続の命令は、ストレージからフェッチまたは再フェッチを行って、直前の命令のコンテキストで実行します。ディレクティブは、**ISYNC** を実行するプロセッサーにのみ影響します。

構文

```
►►──ISYNC──────────────────────────────────────────────────────►◄
```

LIGHT_SYNC

LIGHT_SYNC ディレクティブは、**LIGHT_SYNC** ディレクティブを実行したプロセッサーで新しい命令を実行できるようになる前に、**LIGHT_SYNC** の前のすべての命令が確実に完了するようにします。これにより、 **LIGHT_SYNC** が各プロセッサーからの確認を待たなくなるため、パフォーマンスには最低限の影響しか与えずに複数のプロセッサー間で同期化できます。

構文

```
►►──LIGHT_SYNC─────────────────────────────────────────────────►◄
```

PREFETCH

プリフェッチを使用して、データが参照される前に、データをメイン・メモリーからキャッシュにロードするようコンパイラーに指示することができます。 POWER3 以上のハードウェアによって自動的に行われるプリフェッチもありますが、コンパイラー支援ソフトウェア・プリフェッチは、ソース・コードの中で検出される情報を使用するため、このディレクティブを使用すればキャッシュ・ミスの数を大幅に減らすことができます。

XL Fortran では、コンパイラー支援ソフトウェア・プリフェッチ用に、次の 5 つのディレクティブが提供されています。

- **PREFETCH_BY_LOAD** ディレクティブは、ロード命令によってデータをキャッシュにプリフェッチします。 **PREFETCH_BY_LOAD** はどのマシンでも使用できますが、 POWER3 以上のマシン上で実行している場合は、**PREFETCH_BY_LOAD** を使用するとハードウェア支援プリフェッチを行うことができます。

- **PREFETCH_BY_STREAM** プリフェッチの技法では、 POWER4 プリフェッチ・エンジンを使用して、隣接したキャッシュ・ラインに対する順次アクセスを認識し、次に、メモリー階層のより深いレベルから予期されるラインを要求します。この技法は、メイン・メモリーへの繰り返し参照が行われるに従って、十分なラインがキャッシュにロードされるまで、プリフェッチの深さを増やしながら、パスまたはストリームを確立します。データを減分メモリー・アドレスからフェッチするには、**PREFETCH_BY_STREAM_BACKWARD** ディレクティブを使用します。データを増分メモリー・アドレスからフェッチするには、**PREFETCH_BY_STREAM_FORWARD** ディレクティブを使用します。データをメイン・メモリーからキャッシュにロードするために、このストリーミングされたプリフェッチを使用すると、ロード待ち時間を削減または除去できます。
- **PREFETCH_FOR_LOAD** ディレクティブは、キャッシュ・プリフェッチ命令によって、読み取りのためにデータをキャッシュにプリフェッチします。
- **PREFETCH_FOR_STORE** ディレクティブは、キャッシュ・プリフェッチ命令によって、書き込みのためにデータをキャッシュにプリフェッチします。

構文

PREFETCH ディレクティブには、以下の形式があります。

```
►►—PREFETCH_BY_LOAD—(—prefetch_variable_list—)——————————————————►◄
```

```
►►—PREFETCH_FOR_LOAD—(—prefetch_variable_list—)—————————————————►◄
```

```
►►—PREFETCH_FOR_STORE—(—prefetch_variable_list—)————————————————►◄
```

注: どの PowerPC アーキテクチャーでも有効です。

```
►►—PREFETCH_BY_STREAM_BACKWARD—(—prefetch_variable—)—————————————►◄
```

注: どの PowerPC アーキテクチャーでも有効です。

```
►►—PREFETCH_BY_STREAM_FORWARD—(—prefetch_variable—)——————————————►◄
```

注: どの PowerPC アーキテクチャーでも有効です。

prefetch_variable

プリフェッチされる変数です。変数は、判別可能なストレージ・アドレスを持つデータ・オブジェクトでなければなりません。この変数は、組み込みデータ型および派生データ型を含む、任意のデータ型が可能です。この変数は、プロ

シージャー名、サブルーチン名、モジュール名、関数名、定数、ラベル、ゼロ・サイズのストリング、またはベクトル添え字を持つ配列にすることはできません。

規則

PREFETCH_BY_STREAM_BACKWARD、**PREFETCH_BY_STREAM_FORWARD**、**PREFETCH_FOR_LOAD** および **PREFETCH_FOR_STORE** ディレクティブを使用するには、 PowerPC ハードウェア用にコンパイルする必要があります。

変数をプリフェッチする時は、変数アドレスが入っているメモリー・ブロックがキャッシュにロードされます。メモリー・ブロックは、キャッシュ・ラインのサイズと同じです。キャッシュにロードする変数はメモリー・ブロックのどこにあってもかまわないので、配列のすべてのエレメントをプリフェッチできないこともあります。

これらのディレクティブは、実行可能構造体を入れることのできるソース・コードのどこにあってもかまいません。

これらのディレクティブを使用すると、プログラムの実行時オーバーヘッドが増えることがあります。したがって、ディレクティブを使用するのは必要なときだけにしてください。

プリフェッチ・ディレクティブの効果を最大化するために、単一のプリフェッチの後、または一連のプリフェッチの最後に、 **LIGHT_SYNC** ディレクティブを指定することをお勧めします。

例

例 1: この例は、**PREFETCH_BY_LOAD**、**PREFETCH_FOR_LOAD**、および **PREFETCH_FOR_STORE** ディレクティブの有効な使用法を示しています。

この例は、キャッシュ・ラインのサイズは 64 バイトであり、プログラムの開始時には宣言済みデータ項目はキャッシュに存在していないという前提になっています。ディレクティブを使用することについての理論的解釈は、次のとおりです。

- 配列 *ARRA* のすべてのエレメントが割り当てられるので、 **PREFETCH_FOR_STORE** ディレクティブを使用して、配列の最初の 16 のエレメントと配列の次の 16 のエレメントが参照される前に、これらをキャッシュに入れることができます。
- 配列 *ARRC* のすべてのエレメントが読み取られるので、 **PREFETCH_FOR_LOAD** ディレクティブを使用して、配列の最初の 16 のエレメントと配列の次の 16 のエレメントが参照される前に、これらをキャッシュに入れることができます。 (エレメントは最初に初期化されているという前提になっています。)
- ループのそれぞれの繰り返しでは、変数 *A*、*B*、*C*、*TEMP*、*I*、*K* および配列エレメント *ARRB(I*32)* が使用されているので、 **PREFETCH_BY_LOAD** ディレクティブ

を使用して、変数と配列をキャッシュに入れることができます。 (キャッシュ・ラインのサイズの関係で、エレメント *ARRB(I*32)* から始めて *ARRB* の 16 のエレメントがフェッチされます。)

```
      PROGRAM GOODPREFETCH

      REAL*4 A, B, C, TEMP
      REAL*4 ARRA(2**5), ARRB(2**10), ARRC(2**5)
      INTEGER(4) I, K

! Bring ARRA into cache for writing.
!IBM* PREFETCH_FOR_STORE (ARRA(1), ARRA(2**4+1))

! Bring ARRC into cache for reading.
!IBM* PREFETCH_FOR_LOAD (ARRC(1), ARRC(2**4+1))

! Bring all variables into the cache.
!IBM* PREFETCH_BY_LOAD (A, B, C, TEMP, I , K)

! A subroutine is called to allow clock cycles to pass so that the
! data is loaded into the cache before the data is referenced.
      CALL FOO()
      K = 32
      DO I = 1, 2 ** 5

! Bring ARRB(I*K) into the cache
!IBM* PREFETCH_BY_LOAD (ARRB(I*K))
        A = -I
        B = I + 1
        C = I + 2
        TEMP = SQRT(B*B - 4*A*C)
        ARRA(I) = ARRC(I) + (-B + TEMP) / (2*A)
        ARRB(I*K) = (-B - TEMP) / (2*A)
      END DO
      END PROGRAM GOODPREFETCH
```

例 2: この例は、キャッシュ・ラインのサイズの合計が 256 バイトであり、初期状態では、宣言済みデータ項目はキャッシュまたはレジスターに存在していないことを想定しています。また、配列 *ARRA* および *ARRC* のすべてのエレメントが、キャッシュに読み込まれます。

```
     PROGRAM PREFETCH_STREAM

     REAL*4 A, B, C, TEMP
     REAL*4 ARRA(2**5), ARRC(2**5), ARRB(2**10)
     INTEGER*4 I, K

! All elements of ARRA and ARRC are read into the cache.
!IBM* PREFETCH_BY_STREAM_FORWARD(ARRA(1))
! You can substitute PREFETCH_BY_STREAM_BACKWARD (ARRC(2**5)) to read all
! elements of ARRA and ARRC into the cache.
     K = 32
     DO I = 1, 2**5
        A = -i
```

```
      B = i + 1
      C = i + 2
      TEMP = SQRT(B*B -4*A*C)
      ARRA(I) = ARRC(I) + (-B + TEMP) / (2*A)
      ARRB(I*K) = (-B -TEMP) / (2*A)
   END DO
   END PROGRAM PREFETCH_STREAM
```

関連情報

反復カウントの大きなループへのプリフェッチ技法の適用については、**STREAM_UNROLL** ディレクティブを参照してください。

ハードウェア固有の組み込みプロシージャー

この節では、ハードウェア固有の組み込み関数をアルファベット順に説明します。

FCFI(I)

整数から浮動小数点への変換

浮動小数点変数の整数値を、浮動小数点値に変換します。

この組み込み関数は、64 ビット・モードを使用する 64 ビット PowerPC アーキテクチャーで有効です。

引き数タイプおよび属性

I　タイプは **REAL(8)** でなければなりません。

結果タイプおよび属性

I と同じです。

結果値

I の倍精度浮動小数点値。

例

```
...
REAL*8 :: R8, RES
INTEGER*8 :: I8
EQUIVALENCE(R8, I8)

I8 = 89
RES = FCFI(R8) ! RES = 89.0
...
```

FCTID(X)

浮動小数点から整数への変換

現行の丸めモードを使用して、浮動小数点オペランドを、 64 ビットの符号付き固定小数点整数に変換します。

この組み込み関数は、64 ビット・モードを使用するどの PowerPC アーキテクチャーでも有効です。

引き数タイプおよび属性

X タイプは **REAL(8)** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は、浮動小数点レジスター内の固定小数点整数になります。

FCTIDZ(X)

浮動小数点を整数に変換してゼロに丸める

浮動小数点オペランドを、64 ビットの符号付き固定小数点整数に変換し、ゼロに丸めます。

この組み込み関数は、64 ビットまたは 32 ビット・モードを使用する 64 ビット PowerPC アーキテクチャーで有効です。

引き数タイプおよび属性

X タイプは **REAL(8)** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は、浮動小数点変数内のゼロに丸められた固定小数点整数になります。

FCTIW(X)

浮動小数点から整数への変換

現行の丸めモードを使用して、浮動小数点オペランドを、 32 ビットの符号付き固定小数点整数に変換します。

この組み込み関数は、どの PowerPC または POWER2 アーキテクチャーでも有効です。

引き数タイプおよび属性

X タイプは **REAL(8)** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は、浮動小数点変数内の固定小数点整数になります。

FCTIWZ(X)

浮動小数点を整数に変換してゼロに丸める

浮動小数点オペランドを、32 ビットの符号付き固定小数点整数に変換し、ゼロに丸めます。

この組み込み関数は、どの PowerPC または POWER2 アーキテクチャーでも有効です。

引き数タイプおよび属性

X タイプは **REAL(8)** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は、浮動小数点変数内のゼロに丸められた固定小数点整数になります。

FMADD(A, X, Y)

浮動小数点の乗算加算

浮動小数点の乗算加算の結果を戻します。

引き数タイプおよび属性

A タイプは **REAL(8)** でなければなりません。PowerPC プラットフォームで PowerPC コンパイル用の **-qarch** セットを使用してコンパイルする場合、 *A* は REAL(8) ではなくタイプ **REAL(4)** になる場合があります。

X タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

Y タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

結果タイプおよび属性

A、*X*、および *Y* と同じ。

結果値

結果は *A*X + Y* に等しい値を持ちます。

例

以下の例は、*A*、*B*、*C* および *RES1* が単精度実数であるため、 PowerPC コンパイル用の **-qarch** オプションを使ってコンパイルした場合にのみ有効です。

```
REAL(4) :: A, B, C, RES1
REAL(8) :: D, E, F, RES2

RES1 = FMADD(A, B, C)
RES2 = FMADD(D, E, F)
END
```

FMSUB(A, X, Y)

浮動小数点の乗算減算

浮動小数点の乗算減算の結果を戻します。

引き数タイプおよび属性

A
: タイプは **REAL(8)** でなければなりません。PowerPC プラットフォームで PowerPC コンパイル用の **-qarch** セットを使用してコンパイルする場合、 *A* は REAL(8) ではなくタイプ **REAL(4)** になる場合があります。

X
: タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

Y
: タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

結果タイプおよび属性

A、*X*、および *Y* と同じ。

結果値

結果は *A*X - Y* に等しい値を持ちます。

FNABS(X)

浮動小数点の負の値 -|X| を戻します。

引き数タイプおよび属性

X
: タイプは **REAL** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は、*X* の浮動小数点の負の値 -|X| になります。

例

次の例では、浮動小数点変数の絶対値の内容が負になります。

```
REAL(4) :: A, RES1
REAL(8) :: D, RES2

RES1 = FNABS(A)
RES2 = FNABS(D)
```

FNMADD(A, X, Y)

浮動小数点の負の乗算加算

浮動小数点の負の乗算加算の結果を戻します。

引き数タイプおよび属性

A
: タイプは **REAL(8)** でなければなりません。PowerPC プラットフォームで PowerPC コンパイル用の **-qarch** セットを使用してコンパイルする場合、 *A* は REAL(8) ではなくタイプ **REAL(4)** になる場合があります。

X
: タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

Y
: タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は -(*A***X* + *Y*) に等しい値を持ちます。

FNMSUB(A, X, Y)

浮動小数点の負の乗算減算

浮動小数点の負の乗算減算の結果を戻します。

引き数タイプおよび属性

A
: タイプは **REAL(8)** でなければなりません。PowerPC プラットフォー

ムで PowerPC コンパイル用の **-qarch** セットを使用してコンパイルする場合、 *A* は REAL(8) ではなくタイプ **REAL(4)** になる場合があります。

X タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

Y タイプおよび kind 型付きパラメーターは *A* と同じでなければなりません。

結果タイプおよび属性

A、*X*、および *Y* と同じ。

結果値

結果は -(*A***X* - *Y*) に等しい値を持ちます。

例

以下の例では、**FNMSUB** の結果はタイプ **REAL(4)** です。 この値は **REAL(8)** に変換されてから、*RES* に割り当てられます。

```
REAL(4) :: A, B, C
REAL(8) :: RES

RES = FNMSUB(A, B, C)
END
```

FRES(X)

浮動小数点の逆数演算

浮動小数点の逆数演算の処理結果を戻します。

拡張グラフィックス命令コードを使用するどの PowerPC でも有効です。

引き数タイプおよび属性

X タイプは **REAL(4)** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は 1/*X* の単精度見積もりです。

FRSQRTE(X)

平方根の逆数の処理結果を戻します。

拡張グラフィックス命令コードを使用するどの PowerPC でも有効です。

引き数タイプおよび属性

X タイプは **REAL(8)** でなければなりません。

結果タイプおよび属性

X と同じです。

結果値

結果は *X* の平方根の逆数の倍精度見積もりです。

FSEL(X,Y,Z)

浮動小数点の選択

浮動小数点の選択処理の結果を戻します。この結果は、X の値をゼロと比較することにより判別されます。

拡張グラフィックス命令コードを使用するどの PowerPC でも有効です。

引き数タイプおよび属性

X タイプは **REAL(4)** または **REAL(8)** でなければなりません。

結果タイプおよび属性

X、*Y* および *Z* と同じです。

結果値

- *X* の値がゼロ以上の場合には、*Y* の値が戻されます。
- *X* の値がゼロより小さいかシグナル NaN 値である場合には、 *Z* の値が戻されます。

ゼロ値は符号なしと見なされます。つまり、+0 と -0 は両方ともゼロと等しい値であると見なされます。

MTFSF(MASK, R)

浮動小数点状況フィールドおよび制御レジスター (**FPSCR**) フィールドに移動します。

R の内容は、 *MASK* に指定されたフィールド・マスクの制御のもと、 **FPSCR** に置かれます。

引き数タイプおよび属性

MASK タイプ **INTEGER(4)** のリテラル値でなければなりません。下位 8 ビットが使用されます。

R タイプは **REAL(8)** でなければなりません。

MTFSFI(BF, I)

即時に **FPSCR** フィールドに移動します。

I の値は、*BF* に指定されている **FPSCR** フィールドに置かれます。

引き数タイプおよび属性

BF タイプが **INTEGER(4)** の 0 から 7 のリテラル値でなければなりません。

I タイプが **INTEGER(4)** の 0 から 15 のリテラル値でなければなりません。

MULHY(RA, RB)

高位の乗算

オペランド *RA* と *RB* の 64 ビットまたは 128 ビットの積の、高位の 32 ビットまたは 64 ビットを戻します。

32 ビット整数については、どの PowerPC でも有効です。

64 ビット整数については、64 ビット・モードの 64 ビット・アーキテクチャーを持つ PowerPC で有効です。

引き数タイプおよび属性

RA タイプは整数でなければなりません。

RB タイプは整数でなければなりません。

結果タイプおよび属性

RA、*RB* と同じです。

結果値

オペランド *RA* と *RB* の 32 または 64 ビットの積

ROTATELI(RS, IS, SHIFT, MASK)

即時に左シフトして循環させ MASK を挿入

RS の値を *SHIFT* に指定されたビット数だけ左にシフトして循環させます。次にこの関数は、ビット・マスク *MASK* の下で、*IS* に *RS* を挿入します。

8 バイト整数は 64 ビット・アーキテクチャーでのみ有効です。

引き数タイプおよび属性

RS タイプは整数でなければなりません。

IS タイプは整数でなければなりません。

SHIFT タイプは **INTEGER(4)** でなければなりません。

MASK 整数タイプのリテラル値である必要があります。

結果タイプおよび属性

RS と同じです。

結果値

SHIFT に指定したビット数だけ *RS* を左シフトして循環させ、その結果をビット・マスク *MASK* の下で *IS* に挿入します。

ROTATELM(RS, SHIFT, MASK)

左シフトして循環させマスクと AND 演算

RS の値を *SHIFT* に指定されたビット数だけ左にシフトして循環させます。シフトして循環されたデータは、*MASK* との AND 演算が行われ、結果として戻されます。

引き数タイプおよび属性

RS タイプは整数でなければなりません。8 バイトより小さい整数でなければなりません。

SHIFT タイプは **INTEGER(4)** でなければなりません。

MASK 整数タイプのリテラル値である必要があります。

結果タイプおよび属性

RS と同じです。

結果値

シフトして循環されたデータが、*MASK* と AND 演算されます。

SETFSB0(BT)

0 を FPSCR ビットに移動

FPSCR のビット *BT* が 0 に設定されます。このサブルーチンは値を戻しません。

どの PowerPC でも有効です。

引き数タイプおよび属性

BT タイプは **INTEGER(4)** でなければなりません。

SETFSB1(BT)

1 を FPSCR ビットに移動

FPSCR のビット *BT* が 1 に設定されます。このサブルーチンは値を戻しません。

どの PowerPC でも有効です。

引き数タイプおよび属性

BT タイプは **INTEGER(4)** でなければなりません。

SFTI(M, Y)

浮動小数点を整数に保管

Y の下位 32 ビットの内容が、変換されずに *M* のワードに保管されます。

どの PowerPC でも有効です。

引き数タイプおよび属性

M タイプは **INTEGER(4)** でなければなりません。

Y タイプは **REAL(8)** でなければなりません。

例

```
...
integer*4 :: m
real*8 :: x

x = z"00000000abcd0001"
call sfti(m, x) ! m = z"abcd0001"
..
```

TRAP(A, B, TO)

オペランド *A* がオペランド *B* と比較されます。この比較の結果は、 *TO* と AND 演算されて 5 つの状態になります。結果が 0 以外の場合は、システム・トラップ・ハンドラーが起動されます。

8 バイト整数は 64 ビット・アーキテクチャーでのみ有効です。

引き数タイプおよび属性

A タイプは整数でなければなりません。

B タイプは整数でなければなりません。

TO タイプ **INTEGER(4)** の 1 から 31 までのリテラル値でなければなりません。

IBM 拡張 の終り

第 5 部 付録

付録 A. 異なる標準の間の互換性

この情報は、FORTRAN 77 のユーザーで、Fortran 95、Fortran 90 および XL Fortran については詳しくないユーザーのためのものです。

以下に記載した点以外では、Fortran 90 標準および Fortran 95 標準が、先行の Fortran 国際標準 (Fortran International Standard)、ISO 1539-1:1980 (略式名称 FORTRAN 77) の、上位互換性のある拡張言語です。標準適応の FORTRAN 77 プログラムは、Fortran 90 標準の環境下でも標準適応となります。ただし、組み込みプロシージャーについての下記の項目 4 は例外です。標準適応の FORTRAN 77 プログラムは、削除された機能をプログラムで使用しない限り、 Fortran 95 標準の環境下でも標準適応となります。ただし、組み込みプロシージャーについての下記の項目 4 は例外です。 Fortran 90 標準および Fortran 95 標準では、一部の機能の動作が制限されています (これらの機能は、FORTRAN 77 ではプロセッサー依存のものです)。したがって、これらのプロセッサー依存の機能のうち、いずれかを使用している標準適応の FORTRAN 77 プログラムは、Fortran 90 標準および Fortran 95 標準の環境下で使用すると、標準適応プログラムであるにもかかわらず、変換内容が変わってしまう可能性があります。以下の FORTRAN 77 の機能を Fortran 90 および Fortran 95 の環境下で使用すると、変換結果が異なります。

1. FORTRAN 77 では、**DATA** ステートメントの **DOUBLE PRECISION** データ・オブジェクトの初期化に実定数が使用される場合、プロセッサーは、実際のデータに格納できる定数の精度より高い精度を提供することができました。Fortran 90 および Fortran 95 では、プロセッサーにこのオプションはありません。

   XL Fortran の以前のリリースは、Fortran 90 および Fortran 95 の動作と整合性があります。

2. 共通ブロックに属さない名前付き変数が、**DATA** ステートメントで初期化され、**SAVE** の属性が指定されなかった場合、FORTRAN 77 では、**SAVE** 属性をプロセッサー依存にしました。Fortran 90 標準および Fortran 95 標準では、この名前付き変数に、**SAVE** 属性を持たせるように指定します。

   XL Fortran の以前のリリースは、Fortran 90 および Fortran 95 の動作と整合性があります。

3. FORTRAN 77 では、入力リストで要求する文字数は、入力のフォーマット時にレコードに格納される文字数以下でなければなりませんでした。Fortran 90 標準および Fortran 95 標準では、入力レコードの文字数が不足する場合、適切な **OPEN** ステートメントに **PAD='NO'** 指定子が指定されていなければ、入力レコードに論理的にブランクを埋め込むことを指定します。

   XL Fortran では、**-qxlf77** コンパイラー・オプションの **noblankpad** サブオプションが指定されている場合は、入力レコードにブランクが埋め込まれません。

4. Fortran 90 標準および Fortran 95 標準には、FORTRAN 77 よりも多くの組み込み関数があり、組み込みサブルーチンがいくつか追加されています。したがって、標準適応の FORTRAN 77 プログラムを Fortran 90 および Fortran 95 の環境下で使用すると、変換結果が異なる可能性があります。これは、FORTRAN 77 で、新標準の組み込みプロシージャーのいずれかと同じ名前のプロシージャーを呼び出した場合に起こります。ただし、このプロシージャーを **EXTERNAL** ステートメントに指定した場合は例外です。

   XL Fortran では、指定された名前のプロシージャーが、**-qextern** コンパイラー・オプションによって、**EXTERNAL** ステートメントに指定されたプロシージャーであるかのように扱われます。
5. Fortran 95 では、編集記述子によっては、定様式出力ステートメント内のリスト項目で使う 0 値のフォーマット結果が異なる場合があります。 しかも、Fortran 95 標準は FORTRAN 77 と違い、値を丸めることで出力フィールドの形式にどのような影響を与えるかを指定します。 したがって、値と編集記述子の特定の組み合わせによっては、FORTRAN 77 プロセッサーは Fortran 95 プロセッサーを使用する場合とは異なる出力形式を生成する可能性があります。
6. Fortran 95 では、Fortran 90 では行えなかった、正の実数ゼロと負の実数ゼロをプロセッサーによって見分けることが可能です。 Fortran 95 は、2 番目の引き数が負の実数ゼロの場合、**SIGN** 組み込み関数の動作を変更します。

Fortran 90 の互換性

以下に記載した点以外では、Fortran 95 標準が、先行の Fortran 国際標準 (Fortran International Standard)、ISO/IEC 1539-1:1991 (略式名称 Fortran 90 の、上位互換性のある拡張言語です)。 Fortran 95 標準で削除された機能を使用していない、標準準拠の Fortran 90 プログラムであれば、Fortran 95 プログラムの標準にも準拠しています。 Fortran 95 標準で削除された Fortran 90 の機能は、以下のとおりです。

- **ASSIGN** ステートメントおよび割り当て **GO TO** ステートメント
- **PAUSE** ステートメント
- 実数タイプの **DO** 制御変数および式
- **H** 編集記述子
- **IF** ブロックの外側からの **END IF** ステートメントへの分岐

Fortran 95 では、Fortran 90 では行えなかった、正の実数ゼロと負の実数ゼロをプロセッサーによって見分けることが可能です。 Fortran 95 は、2 番目の引き数が負の実数ゼロの場合、**SIGN** 組み込み関数の動作を変更します。

Fortran 95 標準には、Fortran 90 標準より多くの組み込み関数があります。したがって、標準適応の Fortran 90 プログラムを Fortran 95 標準の環境下で使用すると、変換結果が異なる可能性があります。これは、Fortran 95 で、新標準の組み込みプロシージャーのいずれかと同じ名前のプロシージャーを呼び出した場合に起こります。ただし、

このプロシージャーを **EXTERNAL** ステートメントに指定した場合、およびインターフェース本体を使用して指定した場合は例外です。

使用されなくなった機能

Fortran 言語が進化するにつれて、古い機能のいくつかは、現在のプログラミングの必要に応じて調整され、新しい機能によってさらに効率よく処理されるようになるのはきわめて自然なことです。同時に、受け継がれてきた Fortran コードにかなりの投資がなされたことを考えると、ここで FORTRAN 77 の機能の一部を削除することはユーザーの必要を考慮していないことにもなります。このような理由で、XL Fortran は Fortran 90 標準および FORTRAN 77 標準の完全な上位互換となっています。 Fortran 95 では、Fortran 90 言語標準と FORTRAN 77 言語標準の機能の一部が削除されています。ただし、削除された機能に代わる有効な機能があるので、機能そのものは Fortran 95 から削除されていません。

Fortran 95 は、2 つの旧式の機能のカテゴリーを定義しています。つまり、「削除された機能」および「古くなった機能」です。「削除された機能」とは、Fortran 90 または FORTRAN 77 ではほとんど使用されていないと考えられるため、Fortan 95 ではサポートされなくなった機能です。

「古くなった機能」とは、現在でもよく使用されているものの、それに代わるさらに優れた新規の機能や方式を採用できる FORTRAN 77 の機能です。「古くなった機能」は、定義上では Fortran 95 標準でサポートされていますが、次期の Fortran 標準では「削除された機能」となるものもあります。プロセッサーは「削除された機能」を、言語の拡張機能としてサポートし続ける可能性もありますが、既存のコードを、よりよい方式を採用するように修正していかれることをお勧めします。

Fortran 90 は、以下の FORTRAN 77 機能を「古くなった機能」としています。

- 算術 **IF**

  推奨する方式: 論理 **IF** ステートメント、**IF** 構造体、または **CASE** 構造体を使用してください。

- 実数タイプの **DO** 制御変数および式

  推奨する方式: 整数タイプの変数および式を使用してください。

- **PAUSE** ステートメント

  推奨する方式: **READ** ステートメントを使用してください。

- 選択戻り指定子

  推奨する方式: **CASE** 構造体で戻りコードを評価するか、またはプロシージャーからの戻りで計算 **GO TO** ステートメントを評価してください。

  ```
  ! FORTRAN 77

    CALL SUB(A,B,C,*10,*20,*30)
  ```

```
! Fortran 90

  CALL SUB(A,B,C,RET_CODE)
  SELECT CASE (RET_CODE)
    CASE (1)
     :
    CASE (2)
     :
    CASE (3)
     :
  END SELECT
```

- **ASSIGN** ステートメントおよび割り当て **GO TO** ステートメント

  推奨する方式: 内部プロシージャーを使用してください。
- **IF** ブロックの外側からの **END IF** ステートメントへの分岐

  推奨する方式: **END IF** ステートメントの次のステートメントへの分岐。
- **END DO** または **CONTINUE** 以外での、共用ループ終了およびステートメントでの終了

  推奨する方式: **END DO** または **CONTINUE** ステートメントを使って各ループを終了してください。
- **H** 編集記述子

  推奨する方式: 文字定数編集記述子を使用してください。

Fortran 95 は、以下の FORTRAN 77 機能を「古くなった機能」としています。

- 算術 **IF**

  推奨する方式: 論理 **IF** ステートメント、**IF** 構造体、または **CASE** 構造体を使用してください。
- 選択戻り指定子

  推奨する方式: **CASE** 構造体で戻りコードを評価するか、またはプロシージャーからの戻りで計算 **GO TO** ステートメントを評価してください。

```
! FORTRAN 77

  CALL SUB(A,B,C,*10,*20,*30)

! Fortran 90

  CALL SUB(A,B,C,RET_CODE)
  SELECT CASE (RET_CODE)
    CASE (1)
     :
    CASE (2)
```

```
  ⋮
   CASE (3)

  ⋮
 END SELECT
```

- **END DO** または **CONTINUE** 以外での、共用ループ終了およびステートメントでの終了

  推奨する方式: **END DO** または **CONTINUE** ステートメントを使って各ループを終了してください。
- ステートメント関数
- 実行可能プログラム内の **DATA** ステートメント
- 想定長文字関数
- 固定ソース形式
- 宣言の **CHARACTER*** 形式

削除された機能

Fortran 95 では、以下の Fortran 90 機能および FORTRAN 77 機能を「削除された機能」としています。

- **ASSIGN** ステートメントおよび割り当て **GO TO** ステートメント
- **PAUSE** ステートメント
- 実数タイプの **DO** 制御変数および式
- **H** 編集記述子
- **IF** ブロックの外側からの **END IF** ステートメントへの分岐

付録 B. ASCII 文字セットと EBCDIC 文字セット

XL Fortran では、照合順序として ASCII 文字セットを使用します。

次の表では、標準 ASCII 文字を番号順にリストし、対応する 10 進値と 16 進値をあわせてリストします。 EBCDIC 文字の値を使用するプログラムで作業をする場合のために、EBCDIC 文字に対応する情報も載せてあります。この表では、制御文字を、「`Ctrl-`」の表記を使って記載してあります。たとえば、水平タブ (`HT`) の場合は、「`Ctrl-I`」と記載されています。これは、`Ctrl` キーと `I` キーを同時に押すと入力できます。

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字

10 進値	16 進値	制御文字	ASCII シンボル	意味	EBCDIC シンボル	意味
0	00	Ctrl-@	NUL	ヌル	NUL	ヌル
1	01	Ctrl-A	SOH	見出しの開始	SOH	見出しの開始
2	02	Ctrl-B	STX	テキストの開始	STX	テキストの開始
3	03	Ctrl-C	ETX	テキストの終了	ETX	テキストの終了
4	04	Ctrl-D	EOT	伝送の終了	SEL	選択
5	05	Ctrl-E	ENQ	問い合わせ	HT	水平タブ
6	06	Ctrl-F	ACK	認識	RNL	必須の改行
7	07	Ctrl-G	BEL	ベル	DEL	削除
8	08	Ctrl-H	BS	バックスペース	GE	図形エスケープ
9	09	Ctrl-I	HT	水平タブ	SPS	スーパースクリプト
10	0A	Ctrl-J	LF	改行	RPT	繰り返し
11	0B	Ctrl-K	VT	垂直タブ	VT	垂直タブ
12	0C	Ctrl-L	FF	用紙送り	FF	用紙送り
13	0D	Ctrl-M	CR	改行	CR	改行
14	0E	Ctrl-N	SO	シフトアウト	SO	シフトアウト
15	0F	Ctrl-O	SI	シフトイン	SI	シフトイン
16	10	Ctrl-P	DLE	データ・リンク・エスケープ	DLE	データ・リンク・エスケープ
17	11	Ctrl-Q	DC1	デバイス制御 1	DC1	デバイス制御 1
18	12	Ctrl-R	DC2	デバイス制御 2	DC2	デバイス制御 2
19	13	Ctrl-S	DC3	デバイス制御 3	DC3	デバイス制御 3

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

10 進値	16 進値	制御文字	ASCII シンボル	意味	EBCDIC シンボル	意味
20	14	Ctrl-T	DC4	デバイス制御 4	RES/ENP	復元 / 使用可能表示
21	15	Ctrl-U	NAK	否定応答	NL	改行
22	16	Ctrl-V	SYN	同期信号	BS	バックスペース
23	17	Ctrl-W	ETB	伝送ブロックの終了	POC	プログラム・オペレーター通信
24	18	Ctrl-X	CAN	取り消し	CAN	取り消し
25	19	Ctrl-Y	EM	メディア終端	EM	メディア終端
26	1A	Ctrl-Z	SUB	置換文字	UBS	ユニット・バックスペース
27	1B	Ctrl-[	ESC	エスケープ	CU1	顧客使用 1
28	1C	Ctrl-\	FS	ファイル・セパレーター	IFS	ファイル・セパレーターの交換
29	1D	Ctrl-]	GS	グループ・セパレーター	IGS	グループ・セパレーターの交換
30	1E	Ctrl-∧	RS	レコード・セパレーター	IRS	レコード・セパレーターの交換
31	1F	Ctrl-_	US	ユニット分離	IUS/ITB	ユニット・セパレーターの交換 / 中間伝送ブロック終結
32	20		SP	スペース	DS	数字選択
33	21		!	感嘆符	SOS	重要度の開始
34	22		″	直線二重引用符	FS	フィールド・セパレーター
35	23		#	番号記号	WUS	ワード下線
36	24		$	ドル記号	BYP/INP	バイパス / 禁止表示
37	25		%	パーセント記号	LF	改行
38	26		&	アンパーサンド	ETB	伝送ブロックの終了
39	27		’	アポストロフィ	ESC	escape
40	28		(	左括弧	SA	属性設定
41	29		)	右括弧		
42	2A		*	アスタリスク	SM/SW	モデル・スイッチの設定

表 17. *ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)*

10 進値	**16 進値**	**制御文字**	**ASCII シンボル**	**意味**	**EBCDIC シンボル**	**意味**
43	2B		+	加算記号	CSP	制御列プレフィックス
44	2C		,	コンマ	MFA	フィールド属性の修正
45	2D		-	減算記号	ENQ	問い合わせ
46	2E		.	ピリオド	ACK	認識
47	2F		/	右スラッシュ	BEL	ベル
48	30		0			
49	31		1			
50	32		2		SYN	同期信号
51	33		3		IR	指標戻り
52	34		4		PP	表示位置
53	35		5		TRN	
54	36		6		NBS	数値バックスペース
55	37		7		EOT	伝送の終了
56	38		8		SBS	subscript
57	39		9		IT	インデント・タブ
58	3A		:	コロン	RFF	必須の用紙送り
59	3B		;	セミコロン	CU3	顧客使用 3
60	3C		<	より小	DC4	デバイス制御 4
61	3D		=	等号	NAK	否定応答
62	3E		>	より大		
63						
3F		?	疑問符	SUB	置換文字	
64	40		@	@ 記号	SP	スペース
65	41		A			
66	42		B			
67	43		C			
68	44		D			
69	45		E			
70	46		F			
71	47		G			

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

			ASCII		**EBCDIC**	
10 進値	**16 進値**	**制御文字**	**シンボル**	**意味**	**シンボル**	**意味**
72	48		H			
73	49		I			
74	4A		J		¢	セント
75	4B		K		.	ピリオド
76	4C		L		<	より小
77	4D		M		(	左括弧
78	4E		N		+	加算記号
79	4F		O		|	論理和
80	50		P		&	アンパーサンド
81	51		Q			
82	52		R			
83	53		S			
84	54		T			
85	55		U			
86	56		V			
87	57		W			
88	58		X			
89	59		Y			
90	5A		Z		!	感嘆符
91	5B		[	左大括弧	$	ドル記号
92	5C		\	左スラッシュ	*	アスタリスク
93	5D		]	右大括弧	)	右括弧
94	5E		^	ハット、曲折記号	;	セミコロン
95	5F		_	下線	¬	論理否定
96	60		`	抑音	-	減算記号
97	61		a		/	右スラッシュ
98	62		b			
99	63		c			
100	64		d			
101	65		e			
102	66		f			

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

10 進値	16 進値	制御文字	ASCII シンボル	意味	EBCDIC シンボル	意味
103	67		g			
104	68		h			
105	69		i			
106	6A		j		¦	分割縦線
107	6B		k		,	コンマ
108	6C		l		%	パーセント記号
109	6D		m		_	下線
110	6E		n		>	より大
111	6F		o		?	疑問符
112	70		p			
113	71		q			
114	72		r			
115	73		s			
116	74		t			
117	75		u			
118	76		v			
119	77		w			
120	78		x			
121	79		y		`	抑音
122	7A		z		:	コロン
123	7B		{	左中括弧	#	番号記号
124	7C		|	論理和	@	@ 記号
125	7D		}	右中括弧	'	アポストロフィ
126	7E		~	相似、波形記号	=	等号
127	7F		DEL	削除	"	直線二重引用符
128	80					
129	81				a	
130	82				b	
131	83				c	
132	84				d	
133	85				e	

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

			ASCII		EBCDIC	
10 進値	**16 進値**	**制御文字**	**シンボル**	**意味**	**シンボル**	**意味**
134	86				f	
135	87				g	
136	88				h	
137	89				i	
138	8A					
139	8B					
140	8C					
141	8D					
142	8E					
143	8F					
144	90					
145	91				j	
146	92				k	
147	93				l	
148	94				m	
149	95				n	
150	96				o	
151	97				p	
152	98				q	
153	99				r	
154	9A					
155	9B					
156	9C					
157	9D					
158	9E					
159	9F					
160	A0					
161	A1				~	相似、波形記号
162	A2				s	
163	A3				t	
164	A4				u	

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

			ASCII		EBCDIC	
10 進値	16 進値	制御文字	シンボル	意味	シンボル	意味
165	A5				v	
166	A6				w	
167	A7				x	
168	A8				y	
169	A9				z	
170	AA					
171	AB					
172	AC					
173	AD					
174	AE					
175	AF					
176	B0					
177	B1					
178	B2					
179	B3					
180	B4					
181	B5					
182	B6					
183	B7					
184	B8					
185	B9					
186	BA					
187	BB					
188	BC					
189	BD					
190	BE					
191	BF					
192	C0				{	左中括弧
193	C1				A	
194	C2				B	
195	C3				C	

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

			ASCII		EBCDIC	
10 進値	**16 進値**	**制御文字**	**シンボル**	**意味**	**シンボル**	**意味**
196	C4				D	
197	C5				E	
198	C6				F	
199	C7				G	
200	C8				H	
201	C9				I	
202	CA					
203	CB					
204	CC					
205	CD					
206	CE					
207	CF					
208	D0				}	右中括弧
209	D1				J	
210	D2				K	
211	D3				L	
212	D4				M	
213	D5				N	
214	D6				O	
215	D7				P	
216	D8				Q	
217	D9				R	
218	DA					
219	DB					
220	DC					
221	DD					
222	DE					
223	DF					
224	E0				\	左スラッシュ
225	E1					
226	E2				S	

表 17. ASCII 文字セットと EBCDIC 文字セットの対応文字 (続き)

10 進値	16 進値	制御文字	ASCII シンボル	意味	EBCDIC シンボル	意味
227	E3				T	
228	E4				U	
229	E5				V	
230	E6				W	
231	E7				X	
232	E8				Y	
233	E9				Z	
234	EA					
235	EB					
236	EC					
237	ED					
238	EE					
239	EF					
240	F0				0	
241	F1				1	
242	F2				2	
243	F3				3	
244	F4				4	
245	F5				5	
246	F6				6	
247	F7				7	
248	F8				8	
249	F9				9	
250	FA						縦線
251	FB					
252	FC					
253	FD					
254	FE					
255	FF				EO	eight one (8 個の 1)

特記事項

本書は米国 IBM が提供する製品およびサービスについて作成したものです。

本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、またはサービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品、プログラムの操作またはサービスの評価および検証は、お客様の責任で行っていただきます。

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について実施権を許諾することを意味するものではありません。実施権については、書面にて下記宛先にお送りください。

〒106-0032
東京都港区六本木 3-2-31
IBM World Trade Asia Corporation
Licensing

以下の保証は、国または地域の法律に沿わない場合は、適用されません。

IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更 (たとえば、技術的に不適切な記述や誤植など) は、本書の次版に組み込まれます。 IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムとその他のプログラム（本プログラムを含む）との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、下記に連絡してください。

IBM Corporation
Lab Director
IBM Canada Limited
8200 Warden Avenue
Markham, Ontario, Canada
L6G 1C7

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

IBM の将来の方向または意向に関する記述については、予告なしに変更または撤回される場合があり、単に目標を示しているものです。

本書はプランニング目的としてのみ記述されています。記述内容は製品が使用可能になる前に変更になる場合があります。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

著作権使用許諾:

本書には、さまざまなオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性があることをほのめかしたり、保証することはできません。これらの例は、すべての場合につ

いて完全にテストされたものではありません。 IBM はこれらのプログラムの信頼性、可用性、および機能について法律上の瑕疵担保責任を含むいかなる明示または暗黙の保証責任も負いません。

この情報をソフトコピーでご覧になっている場合は、写真やカラーの図表は表示されない場合があります。

このソフトウェアならびに文書は、その一部をカリフォルニア大学評議員の承諾のもとに提供された「Fourth Berkeley Software Distribution」に基づきます。これの開発にあたった次の研究機関に対し、敬意を表します。: Electrical Engineering and Computer Sciences Department、バークレー・キャンパス

OpenMP は、OpenMP Architecture Review Board の商標です。本書の一部は、 *OpenMP Fortran Language Application Program Interface* バージョン 2.0 (2000 年 11 月) の仕様から転載されたものです。 Copyright 1997-2000 OpenMP Architecture Review Board.

商標

以下は、IBM Corporation の商標です。

AIX	IBM	PowerPC
POWER2 Architecture		

他の会社名、製品名およびサービス名などはそれぞれ各社の商標または登録商標です。

用語集

この用語集では、本書で共通して使用されている用語を定義します。この用語集には、米国規格協会 (ANSI) による定義、および「*IBM コンピューティング辞典* 」から抜粋した項目が含まれています。

［ア行］

暗黙 DO (implied DO). 指標付きの仕様 (**DO** ステートメントと似ているが、**DO** という語の指定はない)。この範囲は、ステートメントのセットではなく、データ・エレメントのリストである。

暗黙インターフェース (implicit interface). プロシージャーそのものからではなく、有効範囲単位から参照されるプロシージャーは、暗黙インターフェースを持つといわれる。ただしこれは、このプロシージャーが、インターフェース・ブロックを持たない外部プロシージャー、インターフェース・ブロックを持たない仮プロシージャー、ステートメント関数のいずれかである場合のみ。

暗黙規定 (predefined convention). 暗黙に指定されたデータ・オブジェクトのタイプと長さの仕様。明示的な仕様が指定されていない場合、名前の最初の文字が基準となる。最初の文字が I 〜 N の場合、長さが 4 の整数タイプとなる。最初の文字が A 〜 H、O 〜 Z、$、_ の場合、長さが 4 の実数タイプとなる。

インターフェース本体 (interface body). **FUNCTION**、**SUBROUTINE** のいずれかのステートメントから、対応する **END** ステートメントまでの、インターフェース・ブロック内のステートメントの順序列。

インターフェース・ブロック (interface block). **INTERFACE** ステートメントから、対応する **END INTERFACE** ステートメントまでのステートメントの順序列。

埋め込まれたブランク (embedded blanks). ブランク以外の文字に囲まれたブランク。

埋め込み (pad). フィールドまたは文字ストリングの未使用の位置を、仮データ (通常は、ゼロまたはブランク) で埋めること。

英字 (alphabetic character). 言語で使用される文字またはその他の記号 (数字を除く)。通常は、英大文字、小文字の A 〜 Z に加え、特定言語で使用可能なその他の特殊記号 (たとえば _ など) を指す。

英数字 (alphanumeric). 文字セットに関するもの。この文字セットには、文字、数字に加え、通常はその他の文字 (たとえば、句読符号、数学記号など) が含まれる。

エレメント型 (elemental). 組み込み演算、プロシージャー、割り当てを修飾する形容詞。修飾されたものは、配列のエレメントまたは規格対応の配列とスカラーのセットと一致したエレメントに対して個別に適用される。

演算子 (operator). Fortran で、1 つまたは 2 つのオペランドが関係する計算の仕様要素。

エンティティー (entity). 次のものを表す一般用語。プログラム単位、プロシージャー、演算子、インターフェース・ブロック、共通ブロック、外部装置、ステートメント関数、タイプ、名前付き変数、式、構成のコンポーネント、名前付き定数、ステートメント・ラベル、構造体、名前リスト・グループなど。

［カ行］

外部プロシージャー (external procedure). 外部サブプログラムまたは Fortran 以外の手段で定義されるプロシージャー。

拡張精度定数 (extended-precision constant). 連続的な 16 バイトのストレージに記憶される実数値に対するプロセッサー近似値。

仮引き数 (dummy argument). 括弧で囲まれたリストに名前が記述されたエンティティー。**FUNCTION**、**SUBROUTINE**、**ENTRY**、ステートメント関数のいずれかのステートメントのプロシージャー名の後ろに存在する。

関係演算子 (relational operator). 関係条件または関係式を表すのに使用される語または記号。

.GT.	より大
.GE.	より大か等しい
.LE.	より小か等しい
.EQ.	等しい
.NE.	等しくない

関係式 (relational expression). 算術式または文字式の次に関係演算子が続き、その次に別の算術式または文字式が続く式。

関数 (function). 単一の変数値を戻すプロシージャー。通常は、単一の出口を持つ。*関数サブプログラム (function subprogram)*、*組み込み関数 (intrinsic function)*、*ステートメント関数 (statement function)* 参照。

キーワード (keyword). (1) ステートメント・キーワードは、ステートメント (またはディレクティブ) の構文の一部の語で、ステートメントを識別するために使用する。 (2) 引き数キーワードは、仮引き数のための名前を指定する。

既存装置 (existing unit). システム特有の有効な装置番号。

共通ブロック (common block). 呼び出し側プログラムと 1 つ以上のサブプログラムによって参照されることのあるストレージ。

区切り文字 (delimiters). 構文のリストを閉じるために使用する括弧またはスラッシュ (あるいはその両方) の組。

組み込み (intrinsic). Fortran 90 で定義され、これ以上の定義や仕様がなくてもどの有効範囲単位内でも使用できるタイプ、演算、割り当てステートメント、プロシージャーを修飾する形容詞。

形式 (format). (1) 文字、フィールド、行などの配置を定義すること。通常は、表示、印刷出力、ファイルなどのために使用される。 (2) 文字、フィールド、行などを配置すること。

継続行 (continuation line). 開始行の次行以降にステートメントを継続させる行。

結果変数 (result variable). 関数の値を戻す変数。

構造体 (construct). **SELECT CASE**、**DO**、**IF**、**WHERE** のいずれかのステートメントで始まり、対応する終端ステートメントで終わるステートメントの順序列。

構造体 (structure). 派生型のスカラー・データ・オブジェクト。

構造体コンポーネント (structure component). そのタイプのコンポーネントに対応する、のデータ・オブジェクトの一部。

コンパイラー・ディレクティブ (compiler directive). ユーザー・プログラムの実行内容ではなく、 XL Fortran の実行内容を制御するソース・コード。

［サ行］

サブオブジェクト (subobject). 名前付きデータ・オブジェクトの一部。ほかの部分とは別々に参照されたり、定義されたりすることがある。配列エレメント、配列セクション、構造体コンポーネント、サブストリングのいずれか。

サブストリング (substring). スカラー文字ストリングの連続する一部分。 (配列セクションでは、サブストリング・セレクターを指定することができるが、結果はサブストリングにはならない。)

サブプログラム (subprogram). 関数サブプログラムまたはサブルーチン・サブプログラム。FORTRAN 77 では、ブロック・データ・プログラム単位は、サブプログラムと呼ばれていたので注意。

サブルーチン (subroutine). **CALL** ステートメントまたは定義された割り当てステートメントから呼び出されるプロシージャー。

算術演算子 (arithmetic operator). 算術演算を実行させる記号。組み込み算術演算子は次のとおり。

+	加算
-	減算
*	乗算
/	除算
**	指数演算

算術式 (arithmetic expression). 1 つ以上の算術演算子と算術 1 次子からなり、結果が単一の数値として表される。算術式は、無符号の算術定数、算術定数の名前、算術変数への参照、関数参照、算術演算子または括弧を使ったこのような 1 次子の組み合わせ。

算術定数 (arithmetic constant). 整数、実数、複素数のいずれかのタイプの定数。

式 (expression). オペランド、演算子、括弧の順序列。変数、定数、関数参照のいずれかを指す場合と、計算を指す場合とがある。

字句エクステント (lexical extent). ディレクティブの字句エクステントには、ディレクティブ構造体内に直接現れるすべてのコードが含まれる。

字句トークン (lexical token). 分割できない固有の解釈を持つ文字の順序列。

事前接続ファイル (preconnected file). 実行可能プログラムの実行時に、最初に装置に接続されるファイル。標準エラー、標準入力、および標準出力はすべて事前接続ファイルである (それぞれ、装置 0、5、6 に接続される)。

実行可能ステートメント (executable statement). プログラムにある処置をとらせるステートメント。たとえば、計算、条件のテスト、通常の順次実行の変更など。

実行可能プログラム (executable program). 自己完結型プロシージャーとして実行できるプログラム。メインプログラムと、オプションで、モジュール、サブプログラム、Fortran 以外の外部プロシージャーからなる。

実行不能ステートメント (nonexecutable statement). プログラム単位、データ、編集情報、ステートメント関数のいずれかの特性を記述するステートメントで、プログラムの実行処理には関係がないもの。

実定数 (real constant). 実数を表す 10 進数のストリング。実定数には、小数点または 10 進指数、あるいはその両方が含まれている。

実引き数 (actual argument). プロシージャー参照で指定される式、変数、プロシージャー、選択戻り指定子のいずれか。

自動並列化 (automatic parallelization). 明示的にコーディングされた **DO** ループと、配列言語用のコンパイラーによって生成されたループの両方をコンパイラーが並列化しようとする場合に行われるプロセスのこと。

順次アクセス (sequential access). ファイル内のレコードの論理順序に従って、ファイルの読み取り、書き込み、除去を行うアクセス方式。

純粋 (pure). 副次作用がないことを示す、プロシージャーの属性。

使用関連付け (use association). 別々の有効範囲単位内での名前の関係。**USE** ステートメントで指定される。

照合順序 (collating sequence). ソート、結合、比較などのためにコンピューター内で順序付けられている文字の順序列。 XL Fortran で使用される AIX の照合順序は、ASCII である。

仕様ステートメント (specification statement). ステートメントのセットの 1 つ。ソース・プログラムで使用されているデータについての情報を提供する。このステートメントは、データ・ストレージを割り振るための情報も提供する。

情報交換用米国標準コード (American National Standard Code for Information Interchange (ASCII)). ANSI で開発されたコード。データ処理システム、データ通信システム、関連する装置間での情報交換に使用される。 ASCII 文字セットは、7 ビットの制御文字と記号文字で構成される。

数字 (digit). 負数ではない整数を表す文字。たとえば、0 ～ 9 のいずれかの数字。

数値定数 (numeric constant). 整数、実数、複素数、バイト数のいずれかを表す定数。

スカラー (scalar). (1) 配列ではない単一のデータ。 (2) 配列となるための特性を持たないもの。

スケール因数 (scale factor). 実数内での小数点の位置を示す番号 (入力の際に、指数がなければ、数の大きさを示す数字)。

ステートメント (statement). 実行処理の順序列または宣言のセット内で、1 つのステップを表す言語構造体。ステートメントには大きく分けて、実行可能と実行不能の 2 つのクラスがある。

ステートメント関数 (statement function). 後ろに仮引き数のリストが続く名前。これは、組み込み式またはの式と等しく、プログラム全体にわたってこれらの式の代わりに使用することができる。

ステートメント・ラベル (statement label). 1 ～ 5 桁の番号。ステートメントの識別に使用される。ステートメント・ラベルは、制御権の移動、**DO** の範囲の定義、**FORMAT** ステートメントへの参照のために使用することができる。

ストレージ関連付け (storage association). 2 つのストレージ順序列間の関係 (ただしこれは、一方の記憶装置がもう一方の記憶装置と同一の場合のみ)。

スレッド (thread). プロセスの集合。その順序によって、実行すべきプロセスが決定される。スレッドはスケジュールされたエレメントであり、タイム・スライスやロック、キューなどのリソースをそれに割り当てることができる。

スレッド可視変数 (thread visible variable). 複数のスレッドから可視である変数。 詳細については、 708 ページの『FLUSH』を参照してください。

正規 (normal). 非正規、無限大、または NaN でない浮動小数点数。

制御ステートメント (control statement). ステートメントの連続的な実行を変更するのに使用されるステートメント。制御ステートメントは、条件ステートメント (**IF** など) の場合と、命令ステートメント (**STOP** など) の場合がある。

整定数 (integer constant). 任意で符号が付けられる数字ストリング。小数点は付けない。

接続装置 (connected unit). XL Fortran では、**OPEN** ステートメントによる名前付きファイルへの明示的接続、暗黙的接続、事前接続といった 3 つの方法のいずれかでファイルに接続された装置。

セレクター (selector). ポインター、ポインティング・デバイス、または選択カーソルのこと。

ゼロ長文字 (zero-length character). 長さが 0 の文字オブジェクト。常に定義される。

ゼロ・サイズ配列 (zero-sized array). 下限を持つ配列。これは、対応する上限より大きい。この配列は、常に定義される。

総称識別子 (generic identifier). **INTERFACE** ステートメントに存在する字句トークン。インターフェース・ブロック内のプロシージャーすべてに関連する。

装置 (unit). I/O ステートメントで使用するためにファイルを参照する手段。装置は、ファイルに接続されるものと接続されないものとがある。接続されている場合には、ファイルを参照する。この接続は対称的である。つまり、装置がファイルに接続されていると、このファイルは装置に接続されていることになる。

添え字 (subscript). 括弧で囲まれた添え字エレメントまたは添え字エレメントのセット。特定の配列エレメントを識別する配列名とともに使用される。

属性 (attribute). データ・オブジェクトの特性。タイプ宣言ステートメント、属性仕様ステートメント、デフォルト設定のいずれかで指定される。

存在 (present). ある仮引き数が実引き数と関連しており、かつ、この実引き数が呼び出しプロシージャーに存在する仮引き数である場合、または呼び出しプロシージャーの仮引き数でない場合、この仮引き数はサブプログラムのインスタンスに存在する。

[タ行]

ターゲット (target). **TARGET** 属性を持つように指定された名前付きのデータ・オブジェクト。ポインター用に **ALLOCATE** ステートメントによって作成されるデータ・オブジェクト、またこのようなオブジェクトのサブオブジェクト。

対称型マルチプロセッシング (Symmetric Multiprocessing (SMP)). マシンのアーキテクチャーに影響を与えることなく、任意のプロセッサーを他のプロセッサーの代わりに使用できる処理。

タイプ宣言ステートメント (type declaration statement). オブジェクトと関数のタイプ、長さ、属性を指定する。オブジェクトには、初期値を割り当てることができる。

タイム・スライス (time slice). タスクを実行するにあたって割り振られた、処理装置での時間間隔。割り振られた時間間隔が経過すると、処理装置時間は別のタスクに割り振られるため、1 つのタスクが一定の制限時間を超えて処理装置を独占することはできない。

チャンク (chunk). 連続するループ反復のサブセット。

注釈 (comment). プログラムにテキストを挿入するための言語構造体。プログラムの実行内容には関係ない。

データ型 (data type). データと関数の特徴を記述する特性および内部表現。組み込みのタイプとしては、整数、実数、複素数、論理、文字の各タイプがある。

データ転送ステートメント (data transfer statement). **READ**、**WRITE**、**PRINT** の各ステートメント。

データ・オブジェクト (data object). 変数、定数、定数のサブオブジェクト。

定義可能 (definable). 割り当てステートメントの左側に名前または指定子を示すことによって、変数の値が変更される可能性がある場合、この変数は定義可能である。

定数 (constant). 不変の値を持つデータ・オブジェクト。変数 と対比。 定数には 4 つのクラスがあり、数字 (算術)、真理値 (論理)、文字データ (文字)、タイプなしのデータ (16 進値、8 進値、2 進値) がこれらに当たる。

定様式データ (formatted data). 指定のフォーマットに従って、主記憶装置と I/O 装置間で転送されるデータ。*リスト指示データ (list-directed data)*、*不定様式データ (unformatted data)* 参照。

ディレクティブ (directive). コンパイラーに指示や情報を与える注釈のタイプ。

適応 (conformance). 実行可能プログラムにおいて Fortran 90 標準に記述されている形式と関係のみを使用している場合、かつ、この実行可能プログラムにおいて Fortran 90 標準に従った解釈を行う場合、このプログラムは Fortran 90 に適応していることになる。実行可能プログラムが標準適応となるようにプログラム単位が実行可能プログラムに含まれている場合、このプログラム単位は Fortran 90 標準に適応している。標準に規定されている解釈を満たすようにプロセッサーが標準適応プログラムを実行する場合、このプロセッサーは標準に適応している。

デバッグ行 (debug line). 固定ソース形式だけに含めることができる、デバッグ用に使うソース・コードを含む行。デバッグ行は、1 桁目の D で定義される。デバッグ行の処理は、**-qdlines** コンパイラー・オプションで制御される。

デフォルトの初期化 (default initialization). 派生型の定義の一部として指定された値を持つオブジェクトの初期化。

トークン (token). プログラム言語で、特定の形式の文字ストリング。定義された重みを持つ。

動的エクステント (dynamic extent). ディレクティブの動的エクステントには、ディレクティブの字句エクステント、および字句エクステント内から呼び出されたすべてのサブプログラムが含まれる。

［ナ行］

名前 (name). 最初が英文字で、その後に 249 文字までの英数字 (英文字、数字、下線) が続く字句トークン。 FORTRAN 77 では、シンボル名と呼ばれていたので注意。

名前付き共通ブロック (named common). 複数個の変数で構成される個別の名前付き共通ブロック。

名前リスト・グループ名 (namelist group name). READ、WRITE、および PRINT ステートメントで使用する名前のリストを指定する NAMELIST ステートメント内の最初のパラメーター。

入出力 (input/output (I/O)). 入力または出力、あるいはその両方に関するもの。

入出力リスト (input/output list). 入力または出力ステートメント内のリスト。読み取りまたは書き込みを行うデータを指定する。出力リストには、定数、演算子、または関数参照を含む式、括弧で囲まれた式のいずれかが含まれることがある。

ネスト (nest). ある種類の 1 つ以上の構造体を、同じ種類の構造体に組み込むこと。たとえば、あるループ (ネストされるループ) を別のループ (ネストするループ) 内にネストしたり、あるサブルーチン (ネストされるサブルーチン) を別のサブルーチン (ネストするサブルーチン) 内にネストしたりする。

［ハ行］

バイト定数 (byte constant). バイト・タイプの名前付き定数。

バイト・タイプ (byte type). 1 バイトのストレージを表すデータ型。**LOGICAL(1)**、**CHARACTER(1)**、**INTEGER(1)** のいずれかを使用できる場合に使用可能。

配列 (array). 順序付けられたスカラー・データのグループを含むエンティティー。配列内のオブジェクトはすべて、同一のデータ型とタイプ・パラメーターを持つ。

配列エレメント (array element). 配列内の単一のデータ項目。配列名と、その後の 1 つ以上の整数式で識別される。この整数式は添え字式と呼ばれ、配列内での位置を指定する。

配列セクション (array section). 配列であり、構造体コンポーネントではないサブオブジェクトのこと。

配列宣言子 (array declarator). ステートメントの一部であり、プログラム単位内で使用される配列について記述するもの。配列宣言子では、配列の名前、含まれる次元数、各次元のサイズを指定する。

配列ポインター (array pointer). 配列を指し示すポインター。

配列名 (array name). 順序付けられたデータ項目のセットの名前。

バインド (bind). 識別子をプログラム内の別のオブジェクトに関係させること。たとえば、識別子を値、アドレス、または別の識別子に関係させること、または仮パラメーターと実パラメーターを関連させることなど。

派生型 (derived type). データがコンポーネントを持つタイプ。各コンポーネントは、組み込みタイプまたは別の派生型のいずれかである。

引き数 (argument). 実引き数または仮引き数。

引き数関連付け (argument association). プロシージャー参照実行時の実引き数と仮引き数の関係。

非既存ファイル (nonexisting file). アクセス可能なストレージ・メディアに物理的には存在しないファイル。

非正規 (denormal). 非常に小さな絶対値と低精度の IEEE 数。非正規数は、ゼロの指数と非ゼロの小数部で表されます。

ファイル (file). レコードの順序列。ファイルが内蔵記憶装置にある場合は、内部ファイルと呼ばれ、 I/O 装置にある場合は、外部ファイルと呼ばれる。

フィールド (field). データの特定のカテゴリーを保管するのに使用されるレコード内の領域。

複素数 (complex number). 順序付けられた 1 対の実数からなる数値。**a+bi** の書式で表される。 **a** および **b** は実数で、**i** の平方は -1 である。

複素数タイプ (complex type). 複素数の値を表すデータ型。この値は順序付けられた 1 対の実数データ項目であり、コンマで区切られ、括弧で囲まれて示される。最初の項目が複素数の実数部で、2 番目の項目が虚数部である。

複素定数 (complex constant). 順序付けられた 1 対の実定数または整定数。コンマで区切られ、括弧で囲まれて示される。最初の定数が複素数の実数部で、2 番目の定数が虚数部である。

不定様式レコード (unformatted record). 内蔵記憶装置と外部記憶装置間で変更されずに伝送されるレコード。

浮動小数点数 (floating-point number). 別々の数値の対で表される実数。最初の数値である小数部と、あらかじめ決められた浮動小数点の基数を 2 番目の数値でべき乗したものとの積。

負のゼロ (negative zero). 指数および小数部が両方ともゼロであるが、符号ビットが 1 である IEEE 表記。負のゼロは正のゼロと等しいとして扱われます。

プログラム単位 (program unit). メインプログラムまたはサブプログラム。

プロシージャー (procedure). プログラムの実行時に呼び出されることのある計算。関数、サブルーチンのいずれか。各プロシージャーは、組み込みプロシージャー、外部プロシージャー、モジュール・プロシージャー、内部プロシージャー、仮プロシージャー、ステートメント関数のいずれかである。サブプログラムに **ENTRY** ステートメントが含まれている場合、サブプログラムでは、1 つ以上のプロシージャーを定義することがある。

ブロック・データ・サブプログラム (block data subprogram). **BLOCK DATA** ステートメントが先頭にあるサブプログラム。名前付き共通ブロックにおいて、変数の初期化に使用される。

米国規格協会 (American National Standards Institute (ANSI)). コンピューターおよび業務装置製造協会 (Computer and Business Equipment Manufacturers Association) 後援の組織。この協会を通じて、正式許可されたいくつかの組織が、独自の業界標準を作成し、保守している。

編集記述子 (edit descriptors). Fortran で、整数、実数、および複素数データを制御する省略形のキーワード。

変数 (variable). 定義可能な値を持つデータ・オブジェクト。この値は、実行可能プログラムの実行時に再定義することができる。名前付きデータ・オブジェクト、配列エレメント、配列セクション、構造体コンポーネント、サブストリングのいずれか。 FORTRAN 77 では、変数は必ずスカラーで、名前が付けられていたことに注意。

ポインター (pointer). **POINTER** の属性を持つ変数。ポインターは、ターゲットに関連するものでなければ、参照したり、定義したりしてはならない。ポインターが配列である場合、関連するポインターでなければ、形状を持たない。

妨害 (interference). **DO** ループ内の 2 つの反復内容が互いに依存しているときの状態を指す。詳細については、 511 ページの『ASSERT』を参照してください。

ホスト (host). 内部プロシージャーを含むメインプログラムまたはサブプログラムは、内部プロシージャーのホストと呼ばれる。モジュール・プロシージャーを含むモジュールは、モジュール・プロシージャーのホストと呼ばれる。

ホスト関連付け (host association). 内部サブプログラム、モジュール・サブプログラム、の定義が、ホストのエンティティーにアクセスするためのプロセス。

ホレリス定数 (Hollerith constant). XL Fortran で表すことのできる文字ストリング。前に **nH** が付く。この n は、文字ストリング内での文字の番号。

［マ行］

マスター・スレッド (master thread). スレッドのグループの主なプロセス。

無限大 (infinity). オーバーフローまたはゼロ割り算で作成された IEEE 数 (正または負)。無限大は、すべてのビットが 1 の指数部とゼロの小数部で表されます。

無名共通ブロック (blank common). 名前のない共通ブロック。

明示的インターフェース (explicit interface). 有効範囲単位内で参照されるプロシージャーのためのもので、内部プロシージャー、モジュール・プロシージャー、組み込みプロシージャー、インターフェース・ブロックを持つ外部プロシージャー、有効範囲単位内の再帰的プロシージャー参照、インターフェース・ブロックを持つ仮プロシージャーのいずれか。

明示的初期化 (explicit initialization). データ・ステートメント初期値リスト、ブロック・データ・プログラム単位、タイプ宣言ステートメント、または配列コンストラクターで宣言された値を持つオブジェクトの初期化。

メインプログラム (main program). プログラムの実行時に最初に制御が渡されるプログラム単位。サブプログラム *(subprogram)* と対比。

文字演算子 (character operator). 文字データに対して実行される演算を表す記号 (たとえば、連結 (//) など)。

文字サブストリング (character substring). 文字ストリングの連続する一部分。

文字式 (character expression). 文字オブジェクト、文字によって評価される関数参照のいずれか。また、連結演算子 (括弧は任意) で分離されるこれらの順序列の場合もある。

文字ストリング (character string). 連続した文字の列。

文字セット (character set). プログラミング言語用またはコンピューター・システム用のすべての有効文字。

文字タイプ (character type). 英数字で構成されるデータ型。データ型 *(data type)* 参照。

文字定数 (character constant). 1 つ以上の英字からなるストリング。アポストロフィまたは二重引用符で囲まれる。

モジュール (module). ほかのプログラム単位からアクセスされる定義を含むプログラム単位、またはこの定義にアクセスするプログラム単位。

戻り指定子 (return specifier). ステートメント (たとえば CALL など) に指定される引き数。RETURN ステートメント内のサブルーチンによって指定されるアクションに応じて、どのステートメント・ラベルに制御を戻すかを示す。

［ヤ行］

有効範囲 (scope). 実行可能プログラムの一部分。この部分では、字句トークン 1 つにつき 1 つの解釈がある。

有効範囲属性 (scope attribute). 実行可能プログラムの一部分。この範囲内では、字句トークンには、特定の指定プロパティーまたはエンティティーの 1 つの解釈が与えられる。

有効範囲単位 (scoping unit). (1) の定義。 (2) インターフェース本体 (ただし、インターフェース本体に含まれるの定義とインターフェース本体は除く)。 (3) プログラム単位またはサブプログラム (ただし、これらに含まれるの定義、インターフェース本体、サブプログラムは除く)。

［ラ行］

ランク (rank). Fortran で、配列の次元の数。

ランダム・アクセス (random access). ファイルの読み取り、書き込み、除去を、任意の順序で行うことができるアクセス方式。

リスト指示 (list-directed). 事前定義の I/O 形式。データ・リスト内のタイプ、タイプ・パラメーター、エンティティーの値に応じて異なる。

リテラル (literal). ソース・プログラム内の記号または数量。データへの参照ではなく、データそのものを指す。

リテラル定数 (literal constant). Fortran で、組み込みタイプのスカラー値を直接表す字句トークン。

ループ (loop). 繰り返し実行されるステートメント・ブロック。

レコード (record). ファイル内でまとめて扱われる値の順序列。

論理演算子 (logical operator). 次のような論理式の演算を表す記号。

.NOT.	(論理否定)
.AND.	(論理積)
.OR.	(論理和)
.EQV.	(論理等価)
.NEQV.	(論理非等価)
.XOR.	(排他的論理和)

論理定数 (logical constant). 真 (`true`) または偽 (`false`) (つまり、T または F) の値を持つ定数。

［ワ行］

割り当てステートメント (assignment statement). 割り当てステートメントは、組み込みの場合と、定義される場合とがある。組み込み割り当てでは、右方オペランドの値を、ストレージ・ロケーションである左方オペランドに記憶する。

［数字］

1 次子 (primary). 式の最も単純な形式。オブジェクト、配列構成子、構造体構成子、関数参照、括弧で囲まれた式のいずれか。

16 進 (hexadecimal). システムに関連する基数が 16 の数字。 16 進数は、0 〜 9 と A (10) 〜 F (15) の範囲にある。

16 進定数 (hexadecimal constant). 通常は、特殊文字で始まる定数。16 進数のみを含む。

2 進定数 (binary constant). 1 つ以上の 2 進数字 (0 と 1) からの定数。

8 進 (octal). システムに関連する基数が 8 の数字。8 進数は、0 〜 7 の範囲にある。

8 進定数 (octal constant). 8 進数からなる定数。

A

ANSI. 米国規格協会 (American National Standards Institute)。

ASCII. 情報交換用米国標準コード (American National Standard Code for Information Interchange)。

D

DO 変数 (DO variable). **DO** ステートメントで指定される変数。 **DO** の範囲内にある 1 つ以上のステートメントの各オカレンスに先立ち、初期化または増分される。範囲内のステートメントの実行回数の制御に使用される。

DO ループ (DO loop). **DO** ステートメントで繰り返し呼び出されるステートメントの範囲。

DOUBLE PRECISION 定数 (DOUBLE PRECISION constant). デフォルトの実際の精度の 2 倍の精度を持つ実数タイプの定数。

F

FORmula TRANslation (Fortran). 高水準プログラミング言語の 1 つ。主に、科学、技術、数学用アプリケーションに使用される。

Fortran. FORmula TRANslation。

I

I/O. 入出力 (Input/output)。

K

kind 型付きパラメーター (kind type parameter). 組み込みタイプの使用可能な種類のラベルを付けたパラメーターの値。

M

mutex. mutex という語は、スレッド間の相互排他 (MUTual EXclusion) を提供するプリミティブ・オブジェクトの短縮語。相互排他 (mutex) は、同時に実行されるスレッドのうち、1 度に 1 つだけがデータへのアクセスを許可されたり特定のアプリケーション・コードを稼働させることを確実にするために、複数のスレッド間で調整的に使用される。

S

SMP.　対称的なマルチプロセッシング (Symmetric Multiprocessing)。

指標

日本語，数字，英字，特殊文字の順に配列されています。なお，濁音と半濁音は清音と同等に扱われています。

［ア行］

アクセス可能性
  パブリック 444
  プライベート 438
アクセスに関する照会 392
値のセパレーター 270
アポストロフィ (') 編集 263
暗黙
  インターフェース 172
  接続 222
  タイプ 69
暗黙 DO
  配列コンストラクターのリスト 103
  DATA ステートメントおよび 329
一般式 113
一般的なサービス・プロシージャーおよび ユーティリティー・プロシージャー 879
因数
  算術 113
  論理 119
インターフェース
  暗黙 172
  ステートメント (INTERFACE) 407
  ブロック 9, 172
インライン注釈 15
英字の定義 943
英数字の定義 943
エスケープ・シーケンス 38
エラー
  回復可能 233
エラー *(続き)*
  重大な 231
  致命的 230
  変換 235
  Fortran 90 言語 237
  Fortran 95 言語 237
エラー条件 230
エレメント型組み込みプロシージャー 539
エレメント型プロシージャー 212
演算
  拡張組み込み 122
  定義済み 122
演算子
  関係 117
  算術 114
  定義済み 179
  の優先順位 123
  文字 116
  論理 120
エンティティー、 有効範囲の 160
エントリー
  関連付け 380
  ステートメント (ENTRY) 359
  名前 359
オブジェクト、データ 28
オプションの引き数 200

［カ行］

外部
  関数 378
  サブプログラム、XL Fortran ライブラリー中の 879
  ファイル 218
改行、$ 編集による回避 263
開始
  値の宣言 327
  行 15
拡張
  組み込み演算 122
拡張 *(続き)*
  精度 (Q) 編集 247
加算算術演算子 114
カプセル化 9
仮プロシージャー 205
仮引き数
  アスタリスク 206
  説明 195
  定義 944
  プロシージャー 205
  変数 201
  INTENT 属性および 199
仮引き数としてのアスタリスク 195, 206
関係
  演算子 117
  式 117
関数
  値 190
  組み込み 879
  サブプログラム 189
  参照 190
  仕様 111
  ステートメント 468
関連付け
  エントリー 380
  共通 316
  使用 166
  整数ポインター 168
  説明 164
  等価 362
  引き数 196
  ポインター 167
  ホスト 164
キーワード
  ステートメント 14
  引き数 194
行
  開始 15
  継続 15
  条件付きコンパイル 22

行 *(続き)*
ソース形式 14
注釈 15
ディレクティブ 507, 695
デバッグ 15, 18
directive 15
共通
関連付け 316
ブロック 12, 314
共通ブロック
データ有効範囲属性文節の中の 750
組み込み
関数
条件付きベクトル・マージ 574
詳細説明 545
総称 192
特定 192
参照： 組み込みプロシージャー
サブルーチン 541
照会
参照： 照会組み込み関数
ステートメント (INTRINSIC) 178
属性 (INTRINSIC) 409
データ型 29
プロシージャー 191
エレメント型 539
サブルーチン 541
照会 539, 541
説明 539, 545
変換 541
INTRINSIC ステートメント内の名前 409
割り当て 127
繰り返しカウント
DATA ステートメントの暗黙 DO リスト内の 329
DO ステートメントおよび 154
繰り返し仕様 373
グローバル・エンティティー 160
計算 GO TO ステートメント 384
形式
形式指示の形式設定 239
形式 *(続き)*
コード 243
固定ソース形式 16
自由ソース形式 19
仕様
文字 377
I/O リストとの相互作用 269
条件付きコンパイル 22
ステートメント (FORMAT) 372
制御 269
IBM 自由ソース形式 21
形式設定
形式指示 239
説明 239
名前リスト 274
リスト指示 270
形状
配列組み込み関数 (SHAPE) 666
配列セクションの 102
配列の 85
継承する長さ
名前付き定数による 311, 487
継続
行 15
文字 16, 21
結果変数 360, 379
言語間呼び出し
%VAL および %REF 関数 197
減算算術演算子 114
構造体
コンストラクター 53
コンポーネント 44
スカラー・コンポーネント 48
制御 8
説明 44
配列コンポーネント 100
CASE 149
DO 152
DO WHILE 157
FORALL 139
IF 147
WHERE 131
構造体エンティティー 160, 164
構造体名 164
固定ソース形式 16
異なる標準の間の互換性 923
コロン (:) 編集 262
コンストラクター
構造体の 53
配列の 102
複素数オブジェクトの 33, 35
コンパイラー・オプション
-I 518
-qalias 197
-qautodbl 545
-qcclines 23
-qci 517
-qctyplss
および CASE ステートメント 305
タイプなし定数 66
文字定数 39, 126
-qddim 88, 435
-qdirective 529
-qdlines 18
-qescape
アポストロフィ編集 264
およびホレリス定数 66
二重引用符編集 264
H 編集および 265
-qextname 879
-qfixed 16
-qintlog 125, 177
-qintsize
組み込みプロシージャーの戻りタイプ 545
整数デフォルトのサイズおよび 29, 36
-qlog4 125
-qmbcs 264, 265
-qmixed 13, 518
-qnoescape 39
-qnosave 79, 391
-qnullterm 39
-qposition 227, 421
-qqcount 259
-qrealsize 31, 545
-qrecur 209
CALL ステートメントおよび 303
ENTRY ステートメント 361

コンパイラー・オプション *(続き)*
-qrecur *(続き)*
FUNCTION ステートメントおよび 381
SUBROUTINE ステートメントおよび 474
-qsave 79, 391
-qsigtrap 669
-qundef 390
-qxflag=oldtab 16
-qxlf77
実数および複素数編集 253, 254
16 進編集 261
2 進編集 246, 251, 260
8 進編集 257
OPEN ステートメントおよび 425
-qxlf90 243, 666
-qzerosize 40
-U 879
コンパイラー・ディレクティブ
参照： ディレクティブ
コンポーネント指定子 49

［サ行］

サービスおよびユーティリティー・サブプログラム
一般 879
説明 879
浮動小数点制御および照会のための効果的なプロシージャー 842
alarm_ 881
bic_ 882
bis_ 882
bit_ 883
clock_ 883
ctime_ 883
date 884
dtime_ 884
etime_ 885
exit_ 885
fdate_ 885
fiosetup_ 886
flush_ 887
サービスおよびユーティリティー・サブプログラム *(続き)*
fpgets および fpsets 841
ftell_ および ftell64_ 887
getarg 888
getcwd_ 888
getfd 889
getgid_ 889
getlog_ 890
getpid_ 890
getuid_ 890
global_timef 891
gmtime_ 892
hostnm_ 892
iargc 893
idate_ 893
ierrno_ 893
irand 894
irtc 894
itime_ 895
jdate 895
lenchr_ 895
lnblnk_ 896
ltime_ 896
mclock 897
qsort_ 897
qsort_down 898
qsort_up 898
rtc 899
setrteopts 899
sleep_ 900
timef 900
timef_delta 901
time_ 900
umask_ 901
usleep_ 902
xl_ _trbk 902
再帰
プロシージャーおよび 209
ENTRY ステートメント 361
FUNCTION ステートメントおよび 380
SUBROUTINE ステートメントおよび 473
最適化、ディレクティブ 510
作業共用構造体
DO / END DO コンパイラー・ディレクティブ 703
SECTIONS / END SECTIONS コンパイラー・ディレクティブ 730
SINGLE / END SINGLE コンパイラー・ディレクティブ 734
サブストリング
範囲
指定 97
配列セクションとの関係 100
文字 40
サブプログラム
外部 169
関数 378
外部 189
内部 189
サービスおよびユーティリティー 879
サブルーチン 189
参照 190
内部 169
呼び出し 169
サブルーチン
関数および 188
組み込み 879
ステートメント (SUBROUTINE) 473
算術
演算子 114
関係式 117
式 113
タイプ
整数 30
複素数 33, 35
実 32
算術 IF ステートメント 386
参照、関数 190
シーケンス派生型 44
時間帯、設定 576
式
一般 113
関係 117
算術 113
次元境界 84

式 *(続き)*
  初期化 110
  仕様 111
  制限 111
  定数 109
  文字 116
  論理 119
  FORMAT ステートメント内の 377
  primary 122
  subscript 94
識別算術演算子 114
字句
  トークン 12
字句エクステントの定義 945
次元境界式 84
指数算術演算子 114
システム照会組み込み関数 541
事前接続 222
実行可能プログラム 169
実行環境ルーチン
  OpenMP 773
実行時オプション
  変更、SETRTEOPTS プロシージャーによる 899
  CNVERR
    変換エラーおよび 235
    READ ステートメントおよび 452
    WRITE ステートメントおよび 505
  ERR_RECOVERY 237
    重大なエラーおよび 231
    変換エラーおよび 235
    BACKSPACE ステートメント 297
    ENDFILE ステートメントおよび 358
    OPEN ステートメントおよび 426
    READ ステートメントおよび 452
    REWIND ステートメントおよび 463
    WRITE ステートメントおよび 505
実行時オプション *(続き)*
  LANGLVL 237, 277
  NAMELIST 281
  NLWIDTH 282
  UNIT_VARS 222, 421
実行順序 25
実数データ型 31
実数編集
  E (指数付き) 247
  F (指数なし) 251
  G (一般) 253
実引き数
  仕様 193
  定義 945
  としてプロシージャー名を指定 366
指定子
  コンポーネントの 49
  配列エレメントの 94
自動オブジェクト 29
自由ソース形式 19
  IBM 21
終端ステートメント 153
順序
  ステートメントの 24
  配列内のエレメントの 95
純粋プロシージャー 209
初期化式 110
照会組み込み関数 539
  BIT_SIZE 562
  DIGITS 580
  EPSILON 585
  HUGE 595
  KIND 609
  LEN 611
  LOC 617
  MAXEXPONENT 625
  MINEXPONENT 631
  PRECISION 646
  PRESENT 647
  RADIX 652
  RANGE 656
  TINY 685
使用関連付け 166, 488
条件付き
  ベクトル・マージ組み込み関数 574
  INCLUDE 518
条件付きコンパイル 22
照合順序 11
乗算算術演算子 114
仕様配列 86
除算算術演算子 114
数字 11
数字ストリング 27
数値編集記述子 243
据え置き形状配列 90
スカラー整数定数名 27
スケール因数 (P) 編集 266
スケジューリング、ブロック巡回 769
ステートメント
  エンティティー 160, 163, 164
  関数ステートメント 468
  終端 153
  順序 24
  説明 13, 285
  ブロック 147
  ラベル 14
  ラベル割り当て (ASSIGN) ステートメント 293
  ラベル・レコード (RECORD) ステートメント 458
  割り当て 127
ストレージ
  共通ブロック内のシーケンス 317
  共用
    共通ブロックの使用 316
    整数ポインターの使用 168
    ポインターの使用 167
    EQUIVALENCE の使用 362
  クラス、変数の
    基本 78
    説明 78
    リテラル 78
    2 次 79
スラッシュ (/) 編集 262
スレッド可視変数 709

- スレッド・セーフ
  - Fortran 90 ポインターの　432
  - pthreads ライブラリー・モジュール　787
- 制御
  - 形式　269
  - 構造体　8
  - ステートメント
    - 計算 GO TO　384
    - 算術 IF　386
    - ブロック IF　387
    - 無条件 GO TO　385
    - 論理 IF　388
    - 割り当て GO TO　382
    - CONTINUE　325
    - DO　335
    - DO WHILE　337
    - END　350
    - PAUSE　430
    - STOP　472
  - 説明　147
  - の転送　25
  - 編集記述子　241, 375
- 制御の転送
  - 説明　25
  - DO ループ内の　155
- 制御マスク　133
- 制限式　111
- 整合性のある配列　105, 540
- 整数
  - データ型　29
  - 編集 (I)　255
  - ポインター関連付け　168
  - POINTER ステートメント　434
- 精度、実オブジェクトの　31
- 切断、ファイルのクローズ　223
- セミコロン・ステートメント・セパレーター　18, 20
- セレクター　12
- ゼロ長のストリング　39
- ゼロ・サイズ配列　84
- 宣言関数　111
- 宣言子
  - 配列　85
  - 範囲指定レベル　160
- 宣言式　111
- 全体配列　83
- 選択戻り
  - 指定子　193, 206
  - 点　195
- ソース形式
  - 固定ソース形式　16
  - 自由ソース形式　19
  - 条件付きコンパイル　22
  - IBM 自由ソース形式　21
- ソース・ファイル・オプション　523, 526
- ソート (qsort_ プロシージャー)　897
- 添え字　94
- 装置、外部ファイル参照　222
- 増分値の処理　155
- 属性
  - 説明　289
  - ALLOCATABLE　289
  - AUTOMATIC　294
  - DIMENSION　334
  - EXTERNAL　366
  - INTENT　405
  - INTRINSIC　409
  - OPTIONAL　427
  - PARAMETER　429
  - POINTER　431
  - PRIVATE　438
  - PROTECTED　442
  - PUBLIC　444
  - SAVE　463
  - STATIC　470
  - TARGET　475
  - VALUE　491
  - VOLATILE　493

［タ行］

- 対称的なマルチプロセッシング
  - 概要　695
  - ディレクティブ　695
- 代替エントリー・ポイント　359
- タイプ宣言　481
  - BYTE　299
  - CHARACTER　306
  - COMPLEX　319
  - DOUBLE COMPLEX　339
- タイプ宣言 *(続き)*
  - DOUBLE PRECISION　342
  - INTEGER　399
  - LOGICAL　411
  - REAL　453
  - TYPE　476
- タイプなし定数
  - 使用　66
  - ホレリス　65
  - 16 進　63
  - 2 進　65
  - 8 進　64
- タイプの決め方　69
- タイプ・パラメーターおよび指定子　27
- 多対 1 セクション　99
- タブ形式設定　16
- 単項演算　107
- チャンク
  - 定義　947
  - SCHEDULE ディレクティブ　726
- 注釈行
  - 固定ソース形式　16
  - 自由形式のソース入力フォーマット　19
  - 説明　15
  - プログラム単位内での順序　25
- 通信、プログラム単位間の
  - 共通ブロックの使用　314
  - 引き数の使用　193
  - モジュールの使用　183
- データ
  - オブジェクト　28
  - ステートメント (DATA)　327
  - タイプ
    - 組み込み　29
    - 事前定義の規則　69
    - 説明　27
    - 派生　41
    - 変換規則　115
  - 編集記述子　239, 244, 373
- データ転送
  - 実行　223
  - ステートメント
    - PRINT　436
    - READ　445

データ転送 *(続き)*
ステートメント *(続き)*
WRITE 500
非同期 225
データ転送ステートメントの実行 223
データ有効範囲属性文節
説明 750
COPYIN 文節 752
定位置 (T、TL、TR、および X) 編集 268
定義状況 70
定義済み演算 122
定義済み演算子 179
定義済み割り当て 180
定数
算術
整数 30
複素数 33, 35
実 32
式 109
説明 28
タイプなし 63
タイプ・パラメーター、指定子と 28
名前付きバイト 126
ホレリス 65
文字 37
論理 36
16 進 63
2 進 65
8 進 64
定様式
レコード 217
INQUIRE ステートメント (FORMATTED) の指定子 392
ディレクティブ
説明 507, 695
ASSERT 511
ATOMIC 697
BARRIER 700
CACHE_ZERO 905
CNCALL 513
COLLAPSE 515
CRITICAL 701
DO SERIAL 707
ディレクティブ *(続き)*
DO (作業共用) 703
EJECT 516
END CRITICAL 701
END DO 703
END MASTER 711
END ORDERED 712
END PARALLEL 715
END PARALLEL DO 718
END PARALLEL SECTIONS 722
END PARALLEL WORKSHARE 726
END SECTIONS 730
FLUSH 708
INCLUDE 517
INDEPENDENT 519
ISYNC 906
LIGHT_SYNC 906
MASTER 711
ORDERED 712
PARALLEL 715
PARALLEL DO 718
PARALLEL SECTIONS 722
PARALLEL WORKSHARE 726
PERMUTATION 525
PREFETCH_BY_LOAD 906
PREFETCH_FOR_LOAD 906
PREFETCH_FOR_STORE 906
SCHEDULE 726
SECTIONS 730
SINGLE / END SINGLE 734
SNAPSHOT 527
SOURCEFORM 529
STREAM_UNROLL 530
SUBSCRIPTORDER 532
THREADLOCAL 738
THREADPRIVATE 741
UNROLL 534
UNROLL_AND_FUSE 536
WORKSHARE 747
#line 523
@PROCESS 526
ディレクティブ行 15
ディレクティブ文節、グローバル規則 750
デバッグ行 15, 18
デフォルト・タイプ 69
等価
論理 120
動的エクステントの定義 948
特殊文字 11
ドル記号 ($) 編集 263

[ナ行]

内部
関数 378
プロシージャー 169
名前
エントリー 359
説明 12
総称または特定関数の 192
タイプの決め方 69
判別、ストレージ・クラスの 78
有効範囲 160
名前付き共通ブロック 316
名前付き定数から継承される長さ 311, 487
名前付き定数バイト 126
名前リスト
グループ 12
形式設定 274
二重引用符 (″″) 編集 263

[ハ行]

排他的論理和、論理 120
倍精度 (D) 編集 247
配列
エクステント 84
エレメント 94
境界 84
形状 85
コンストラクター 102
サイズ 85
自動 87
仕様の 86
据え置き形状 90
整合 88
セクション 95
説明 83

配列 *(続き)*
ゼロ・サイズの　84
宣言子　85
想定形状　89
想定サイズ　92, 111
配列ポインター　91
ポインター　91
ポインティング先　88
明示的形状　86
ランク　85
割り振り可能　90
配列の次元　84
派生型
決め方、タイプの　47
構造体コンストラクター　53
構造体コンポーネント　44
スカラー構造体コンポーネント　48
説明　41
配列構造体コンポーネント　100
派生型ステートメント　333
引き数
キーワード　194
仕様　193
定義　949
否定
算術演算子　114
論理演算子　120
非等価、論理　120
非同期 I/O
データ転送および　225
INQUIRE ステートメントおよび　393
OPEN ステートメントおよび　421
READ ステートメントおよび　448
WAIT ステートメントおよび　496
WRITE ステートメントおよび　503
ファイル、外部　218
ファイル位置
データ転送の前後の　227
BACKSPACE ステートメント、実行の後の　297
ファイル位置 *(続き)*
ENDFILE ステートメント、実行の後の　358
REWIND ステートメント、実行の後の　462
ファイル位置付けステートメント
BACKSPACE ステートメント　296
ENDFILE ステートメント　357
REWIND ステートメント　461
ファイル終了レコード　218
ファイルの終わり条件　230
フィールド編集　243
複素数
データ型　35
編集　244
符号制御 (S、SS、および SP) 編集　267
不定様式レコード　218
浮動小数点制御および照会のための効果的なプロシージャー
説明　842
clr_fpscr_flags　845
fp_trap　846
get_fpscr　846
get_fpscr_flags　847
get_round_mode　847
set_fpscr　848
set_fpscr_flags　848
set_round_mode　849
ブランク
共通ブロック　316
形式設定時の解釈、設定　264
指定子
INQUIRE ステートメント (BLANK) の　392
OPEN ステートメント (BLANK) の　421
ゼロ (BZ) 編集　264
ヌル (BN) 編集　264
編集　264
プログラム単位　169
プロシージャー
仮　205
外部　169, 441
内部　169
プロシージャー、サブプログラムによって呼び出される　169
プロシージャー参照　190
ブロック
巡回スケジューリング　769
ステートメント　147
ELSE　148
ELSE IF　148
IF　148, 387
ブロック・データ
ステートメント (BLOCK DATA)　298
プログラム単位　187
分岐、制御の　157
ベクトル添え字　99
変換組み込み関数　541
編集
説明　243
複素数　244
文字カウント Q　259
文字ストリング　263
A (文字)　244
B (2 進)　245
BN (ブランク・ヌル)　264
BZ (ブランク・ゼロ)　264
D (倍精度)　247
E (指数付き実数)　247
EN　249
ES　250
F (指数なし実数)　251
G (一般)　253
H　265
I (整数)　255
L (論理)　256
O (8 進)　257
P (スケール因数)　266
Q (拡張精度)　247
S、SS、および SP (符号制御)　267
T、TL、TR、および X (定位置)　268
Z (16 進)　260
" (二重引用符)　263
$ (ドル記号)　263
' (アポストロフィ)　263
/ (スラッシュ)　262

編集 *(続き)*
: (コロン) 262
編集記述子
数値 243
制御 (繰り返し不能) 241, 375
データ (繰り返し可能) 239, 373
名前 12
文字ストリング 242, 376
変数
形式の式 377
説明 28
変数のサブオブジェクト 28
ポインター
関連付け 167
属性、POINTER (Fortran 90) 431
割り当て 142
ポインティング先
配列 88
妨害 512, 519
包括的論理和、論理 120
ホスト
関連付け 159, 164
有効範囲単位 159
保留制御マスク 133
ホレリス定数 12, 65
ホワイト・スペース 12

[マ行]

丸めモード 115
マクロ、_OPENMP C プリプロセッサー 23
マスクされた ELSEWHERE ステートメント 131, 348
マスクされた配列割り当て 133
マルチバイト文字 39
右マージン 16
無限大
数値出力編集での指示方法 248
無条件 GO TO ステートメント 385
明示的
インターフェース 172
タイプ 69
明示的形状配列 86
明白な参照 176
メインプログラム 181, 441
文字 11
演算子 116
関係式 118
形式仕様 377
サブストリング 40
式 116
ストリング編集記述子 242, 376
セット 11
編集
(A) 244
(Q)、カウント 259
マルチバイト 39
文字ストリング編集 263
モジュール
参照 166, 488
ステートメント (MODULE) 416
説明 9, 183
戻り指定子 25
戻り点および指定子、代替 193

[ヤ行]

有効範囲
データ有効範囲属性文節 750
有効範囲、エンティティーおよび 160
有効範囲単位 159
優先順位
算術演算子の 114
すべての演算子の 123
論理演算子の 120
呼び出しコマンド 15

[ラ行]

ライブラリー・サブプログラム 879
ラベル、ステートメント 14
ランク
配列セクションの 102
配列の 85
リスト指示の形式設定 270
リテラル・ストレージ・クラス 78
リンカー・オプション
-brename 879
ループ
制御の処理 154
ループ *(続き)*
ループによる依存性 512, 519
DO 構造体および 152
レコード
ステートメント
ステートメント・ラベル (RECORD) 458
説明 217
定様式 217
ファイル終了 218
不定様式 218
レコードの終わり、$ 編集による回避 263
レコードの終わり条件 230
連結演算子 116
ローカル・エンティティー 160, 161
ロック・ルーチン
OpenMP 773
論理
組み込み関数 (LOGICAL) 619
式 119
タイプ宣言ステートメント (LOGICAL) 411
データ型 36
等価 120
排他的論理和 120
否定 120
非等価 120
包括的論理和 120
論理積 120
IF ステートメント 388
(L) 編集 256
論理和、論理 119, 120
論理積、論理 120

[ワ行]

割り当て
組み込み 127
ステートメント
ステートメント・ラベル (ASSIGN) 293
説明 127
定義済み 180
ポインター 142
マスクされた配列 133

割り当て GO TO ステートメント 382
割り振り状況 77

[数字]

1 次子 122
1 次子 (式) 108
16 進
　定数 63
　(Z) 編集 260
2 進
　演算 107
　定数 65
　編集 (B) 245
8 進 (O) 編集 257
8 進定数 64

A

A (文字) 編集 244
ABORT 組み込みサブルーチン 545
ABS
　組み込み関数 546
　初期化式 110
　特定名 547
ACCESS 指定子
　INQUIRE ステートメントの 392
　OPEN ステートメントの 421
ACHAR 組み込み関数 547
ACOS
　組み込み関数 547
　特定名 548
ACOSD
　組み込み関数 548
　特定名 549
ACTION 指定子
　INQUIRE ステートメントの 392
　OPEN ステートメントの 421
ADJUSTL 組み込み関数 549
ADJUSTR 組み込み関数 549
ADVANCE 指定子
　READ ステートメントの 446
　WRITE ステートメントの 501
AIMAG
　組み込み関数 550
AIMAG *(続き)*
　初期化式 110
　特定名 551
AINT
　組み込み関数 551
　特定名 552
AIX Pthreads ライブラリー 787
alarm_ サービスおよびユーティリティー・サブプログラム 881
ALGAMA 特定名 613
ALL 配列組み込み関数 552
ALLOCATABLE 属性 289
ALLOCATE ステートメント 290
ALLOCATED 配列組み込み関数 292, 553
ALOG 特定名 618
ALOG10 特定名 619
AMAX0 特定名 624
AMAX1 特定名 624
AMIN0 特定名 631
AMIN1 特定名 631
AMOD 特定名 636
AND 特定名 597
AND 論理演算子 120
ANINT 組み込み関数 553
ANINT 特定名 554
ANSI の定義 950
ANY 配列組み込み関数 554
asa コマンド 217
ASCII
　定義 946
　文字セット 11, 929
ASIN
　組み込み関数 555
　特定名 556
ASIND
　組み込み関数 556
　特定名 557
ASSERT コンパイラー・ディレクティブ 511
ASSIGN ステートメント 293
ASSOCIATED 組み込み関数 292, 557
ASYNCH 指定子
　INQUIRE ステートメントの 392
　OPEN ステートメントの 421
ATAN
　組み込み関数 558
　特定名 559
ATAN2
　組み込み関数 559
　特定名 561
ATAN2D
　組み込み関数 561
　特定名 562
ATAND
　組み込み関数 559
　特定名 559
ATOMIC コンパイラー・ディレクティブ 697
AUTOMATIC 属性 294

B

B (2 進) 編集 245
BACKSPACE ステートメント 296
BARRIER コンパイラー・ディレクティブ 700
bic_ サービスおよびユーティリティー・サブプログラム 882
bis_ サービスおよびユーティリティー・サブプログラム 882
bit_ サービスおよびユーティリティー・サブプログラム 883
BIT_SIZE
　組み込み、定数式 109
　組み込み関数 111, 562
BN (ブランク・ヌル) 編集 264
BTEST
　組み込み関数 563
　特定名 564
BYTE タイプ宣言ステートメント 299
BZ (ブランク・ゼロ) 編集 264

C

CABS 特定名 547
CACHE_ZERO コンパイラー・ディレクティブ 905
CALL ステートメント 302

CASE
　構造体　149, 304
　ステートメント　304
CCOS 特定名　568
CDABS 特定名　547
CDCOS 特定名　568
CDEXP 特定名　588
CDLOG 特定名　618
CDSIN 特定名　669
CDSQRT 特定名　677
CEILING 組み込み関数　564
CEXP 特定名　588
CHAR
　組み込み関数　565
　特定名　566
CHARACTER タイプ宣言ステートメント　306
chtz コマンド　576
clock_ サービスおよびユーティリティー・サブプログラム　883
CLOG 特定名　618
CLOSE ステートメント　312
clr_fpscr_flags サブプログラム　845
CMPLX
　組み込み関数　566
　初期化式　110
　特定名　567
CNCALL コンパイラー・ディレクティブ　513
CNVERR 実行時オプション
　暗黙 DO リストおよび　452, 505
　変換エラーおよび　235
COLLAPSE コンパイラー・ディレクティブ　515
COMMON ステートメント　314
COMPLEX タイプ宣言ステートメント　319
CONJG
　組み込み関数　567
　初期化式　110
　特定名　568
CONTAINS ステートメント　324
CONTINUE ステートメント　325
COPYIN 文節　752
COS
　組み込み関数　568
COS *(続き)*
　特定名　568
COSD
　組み込み関数　569
　特定名　569
COSH
　組み込み関数　569
　特定名　570
COUNT 配列組み込み関数　570
CPU_TIME 組み込み関数　571
cpu_time_type 実行時オプション　571
CQABS 特定名　547
CQCOS 特定名　568
CQEXP 特定名　588
CQLOG 特定名　618
CQSIN 特定名　669
CQSQRT 特定名　677
CRITICAL コンパイラー・ディレクティブ　701
CSHIFT 配列組み込み関数　573
CSIN 特定名　669
CSQRT 特定名　677
ctime_ サービスおよびユーティリティー・サブプログラム　883
CVMGM、 CVMGN、 CVMGP、 CVMGT、 CVMGZ 組み込み関数　574
CYCLE ステートメント　325

D

D デバッグ行　15
D (倍精度) 編集　247
DABS 特定名　547
DACOS 特定名　548
DACOSD 特定名　549
DASIN 特定名　556
DASIND 特定名　557
DATAN 特定名　559
DATAN2 特定名　561
DATAN2D 特定名　562
DATAND 特定名　559
date サービスおよびユーティリティー・サブプログラム　884
DATE_AND_TIME 組み込みサブルーチン　576
DBLE
　組み込み関数　578
　初期化式　110
　特定名　579
DBLEQ 特定名　579
DCMPLX
　組み込み関数　579
　初期化式　110
　特定名　580
DCONJG 特定名　568
DCOS 特定名　568
DCOSD 特定名　569
DCOSH 特定名　570
DDIM 特定名　581
DEALLOCATE ステートメント　331
DELIM 指定子
　INQUIRE ステートメントの　392
　OPEN ステートメントの　421
DERF 特定名　586
DERFC 特定名　587
DEXP 特定名　588
DFLOAT 特定名　579
DIGITS 組み込み関数　580
DIM
　組み込み関数　581
　初期化式　110
　特定名　581
DIMAG 特定名　551
DIMENSION 属性　334
DINT 特定名　552
DIRECT 指定子、INQUIRE ステートメントの　392
DLGAMA 特定名　613
DLOG 特定名　618
DLOG10 特定名　619
DMAX1 特定名　624
DMIN1 特定名　631
DMOD 特定名　636
DNINT 特定名　554
DO
　ステートメント　152, 335
　ループ　152, 336
DO SERIAL コンパイラー・ディレクティブ　707

DO WHILE
構造体 157
ステートメント 337
ループ 338
DO (作業共用) コンパイラー・ディレクティブ
説明 703
SCHEDULE 文節 703
DONE 指定子、WAIT ステートメントの 496
DOT_PRODUCT 配列組み込み関数 582
DOUBLE COMPLEX タイプ宣言ステートメント 339
DOUBLE PRECISION タイプ宣言ステートメント 342
DPROD
組み込み関数 582
初期化式 110
特定名 583
DREAL 特定名 658
DSIGN 特定名 667
DSIN 特定名 669
DSIND 特定名 670
DSINH 特定名 671
DSQRT 特定名 677
DTAN 特定名 683
DTAND 特定名 684
DTANH 特定名 684
dtime_ サービスおよびユーティリティー・サブプログラム 884

E

E (指数付き実数) 編集 247
EBCDIC 文字セット 929
EJECT コンパイラー・ディレクティブ 516
ELEMENTAL 212
ELSE
ステートメント 148, 346
ブロック 148
ELSE IF
ステートメント 148, 347
ブロック 148
ELSEWHERE ステートメント 131, 348
EN 編集 249
END CRITICAL コンパイラー・ディレクティブ 701
END DO コンパイラー・ディレクティブ 703
END DO ステートメント 152, 351
END FORALL ステートメント 351
END IF ステートメント 148, 351
END INTERFACE ステートメント 172, 354
END MASTER コンパイラー・ディレクティブ 711
END ORDERED コンパイラー・ディレクティブ 712
END PARALLEL DO コンパイラー・ディレクティブ 718
END PARALLEL SECTIONS コンパイラー・ディレクティブ 722
END PARALLEL WORKSHARE コンパイラー・ディレクティブ 726
END PARALLEL コンパイラー・ディレクティブ 715
END SECTIONS コンパイラー・ディレクティブ 730
END SELECT ステートメント 351
END TYPE ステートメント 356
END WHERE ステートメント 131, 351
END 指定
READ ステートメントの 446
WAIT ステートメントの 496
END ステートメント 350
ENDFILE ステートメント 357
EOR 指定子、READ ステートメントの 446
EOSHIFT 配列組み込み関数 583
EPSILON 組み込み関数 585
EQUIVALENCE
関連付け 362
COMMON での制約事項 317
EQUIVALENCE ステートメント 362
EQV 論理演算子 120
ERF
組み込み関数 586
特定名 586
ERFC
組み込み関数 587
特定名 587
ERR 指定子
BACKSPACE ステートメントの 296
CLOSE ステートメントの 313
ENDFILE ステートメントの 357
INQUIRE ステートメントの 392
OPEN ステートメントの 421
READ ステートメントの 446
REWIND ステートメントの 462
WAIT ステートメントの 496
WRITE ステートメントの 501
ERR_RECOVERY 実行時オプション
重大なエラーおよび 231
変換エラーおよび 235
BACKSPACE ステートメント 297
EDNFILE ステートメントおよび 358
Fortran 90 言語エラーおよび 237
Fortran 95 言語エラーおよび 237
OPEN ステートメントおよび 426
READ ステートメントおよび 452
REWIND ステートメントおよび 463
WRITE ステートメントおよび 505
ES 編集 250
etime_ サービスおよびユーティリティー・サブプログラム 885
execution_part 182
EXIST 指定子、INQUIRE ステートメントの 392
EXIT ステートメント 365
exit_ サービスおよびユーティリティー・サブプログラム 885

EXP
組み込み関数 588
特定名 588
EXPONENT 組み込み関数 589
EXTERNAL 属性 366

F

F (指数なし実数) 編集 251
FCFI PowerPC 組み込み関数 910
FCTID PowerPC 組み込み関数 910
FCTIDZ PowerPC 組み込み関数 911
FCTIW PowerPC 組み込み関数 911
FCTIWZ PowerPC 組み込み関数 912
fdate_ サービスおよびユーティリティー・サブプログラム 885
fexcp.h インクルード・ファイル 668
FILE 指定子
INQUIRE ステートメントの 392
OPEN ステートメントの 421
fiosetup_ サービスおよびユーティリティー・サブプログラム 886
FLOAT 特定名 658
FLOOR 組み込み関数 589
FLUSH コンパイラー・ディレクティブ 708
flush_ サービスおよびユーティリティー・サブプログラム 887
FMADD PowerPC 組み込み関数 912
FMSUB PowerPC 組み込み関数 913
FMT 指定子
PRINT ステートメントの 436
READ ステートメントの 446
WRITE ステートメントの 501
FNABS PowerPC 組み込み関数 913
FNMADD PowerPC 組み込み関数 914
FNMSUB PowerPC 組み込み関数 914
FORALL
構造体 139
ステートメント 367
FORALL (構造体) ステートメント 371
FORM 指定子
INQUIRE ステートメントの 392
OPEN ステートメントの 421
fpgets および fpsets サービスおよびユーティリティー・サブプログラム 841
fpscr の定数
一般 843
リスト 843
例外の詳細フラグ 844
例外の要約フラグ 844
fp_trap の定数 845
IEEE 丸めモード 843
IEEE 例外使用可能フラグ 844
IEEE 例外状況フラグ 844
fpscr プロシージャー
説明 842
clr_fpscr_flags 845
fp_trap 846
get_fpscr 846
get_fpscr_flags 847
get_round_mode 847
set_fpscr 848
set_fpscr_flags 848
set_round_mode 849
fp_trap サブプログラム 846
FRACTION 組み込み関数 591
FRE PowerPC 組み込み関数 915
FRSQRTE PowerPC 組み込み関数 915
FSEL PowerPC 組み込み関数 916
ftell_ および ftell64_ サービスおよびユーティリティー・サブプログラム 887
FUNCTION ステートメント 378
f_maketime 関数 790
f_pthread 787
f_pthread_attr_destroy 関数 790
f_pthread_attr_getdetachstate 関数 791
f_pthread_attr_getguardsize 関数 791
f_pthread_attr_getinheritsched 関数 792
f_pthread_attr_getschedparam 関数 792
f_pthread_attr_getschedpolicy 関数 793
f_pthread_attr_getscope 関数 793
f_pthread_attr_getstackaddr 関数 794
f_pthread_attr_getstacksize 関数 795
f_pthread_attr_init 関数 795
f_pthread_attr_setdetachstate 関数 796
f_pthread_attr_setguardsize 796
f_pthread_attr_setinheritsched 関数 797
f_pthread_attr_setschedparam 関数 798
f_pthread_attr_setschedpolicy 関数 798
f_pthread_attr_setscope 関数 799
f_pthread_attr_setstackaddr 関数 800
f_pthread_attr_setstacksize 関数 800
f_pthread_attr_t 関数 801
f_pthread_cancel 関数 801
f_pthread_cleanup_pop 関数 801
f_pthread_cleanup_push 関数 802
f_pthread_condattr_destroy 関数 807
f_pthread_condattr_getpshared 関数 807
f_pthread_condattr_init 関数 808
f_pthread_condattr_setpshared 関数 808
f_pthread_condattr_t 関数 809
f_pthread_cond_broadcast 関数 803
f_pthread_cond_destroy 関数 804
f_pthread_cond_init 関数 804
f_pthread_cond_signal 関数 805
f_pthread_cond_t 関数 805
f_pthread_cond_timedwait 関数 805
f_pthread_cond_wait 関数 806
f_pthread_create 関数 809
f_pthread_detach 関数 810
f_pthread_equal 関数 811
f_pthread_exit 関数 811
f_pthread_getconcurrency 関数 812
f_pthread_getschedparam 関数 812
f_pthread_getspecific 関数 813
f_pthread_join 関数 813
f_pthread_key_create 関数 814
f_pthread_key_delete 関数 814
f_pthread_key_t 815

f_pthread_kill 関数 815
f_pthread_mutexattr_destroy 関数 819
f_pthread_mutexattr_getprioceiling 関数 820
f_pthread_mutexattr_getprotocol 関数 820
f_pthread_mutexattr_getpshared 関数 820
f_pthread_mutexattr_gettype 関数 821
f_pthread_mutexattr_init 関数 822
f_pthread_mutexattr_setprioceiling 関数 822
f_pthread_mutexattr_setprotocol 関数 823
f_pthread_mutexattr_setpshared 関数 823
f_pthread_mutexattr_settype 関数 824
f_pthread_mutexattr_t 825
f_pthread_mutex_destroy 関数 816
f_pthread_mutex_getprioceiling 関数 816
f_pthread_mutex_init 関数 816
f_pthread_mutex_lock 関数 817
f_pthread_mutex_setprioceiling 関数 818
f_pthread_mutex_t 818
f_pthread_mutex_trylock 関数 818
f_pthread_mutex_unlock 関数 819
f_pthread_once 関数 825
f_pthread_once_t 826
f_pthread_rwlockattr_destroy 関数 831
f_pthread_rwlockattr_getpshared 関数 831
f_pthread_rwlockattr_init 関数 832
f_pthread_rwlockattr_setpshared 関数 832
f_pthread_rwlockattr_t 関数 833
f_pthread_rwlock_destroy 関数 826
f_pthread_rwlock_init 関数 826
f_pthread_rwlock_rdlock 関数 827
f_pthread_rwlock_t 関数 828
f_pthread_rwlock_tryrdlock 関数 828
f_pthread_rwlock_trywrlock 関数 829
f_pthread_rwlock_unlock 関数 830
f_pthread_rwlock_wrlock 関数 830
f_pthread_self 関数 833
f_pthread_setcancelstate 関数 833
f_pthread_setcanceltype 関数 834
f_pthread_setconcurrency 関数 834
f_pthread_setschedparam 関数 835
f_pthread_setspecific 関数 836
f_pthread_t 関数 836
f_pthread_testcancel 関数 837
f_sched_param 関数 837
f_sched_yield 関数 837
f_timespec 関数 837

G

G (一般) 編集 253
GAMMA
  組み込み関数 592
  特定名 592
getarg サービスおよびユーティリティー・サブプログラム 888
getcwd_ サービスおよびユーティリティー・サブプログラム 888
GETENV 組み込みサブルーチン 593
getfd サービスおよびユーティリティー・サブプログラム 889
getgid_ サービスおよびユーティリティー・サブプログラム 889
getlog_ サービスおよびユーティリティー・サブプログラム 890
getpid_ サービスおよびユーティリティー・サブプログラム 890
getuid_ サービスおよびユーティリティー・サブプログラム 890
get_round_mode サブプログラム 847
get__fpscr サブプログラム 846
get__fpscr_flags サブプログラム 847
global_timef サービスおよびユーティリティー・サブプログラム 891
gmtime_ サービスおよびユーティリティー・サブプログラム 892
GO TO ステートメント
  計算 384
  無条件 385
  割り当てられた 382

H

H 編集 265
HFIX エレメント型関数 594
HFIX 特定名 594
hostnm_ サービスおよびユーティリティー・サブプログラム 892
HUGE 組み込み関数 595

I

I (整数) 編集 255
IABS 特定名 547
IACHAR 組み込み関数 595
IAND
  組み込み関数 596
  特定名 597
iargc サービスおよびユーティリティー・サブプログラム 893
IBCLR
  組み込み関数 597
  特定名 598
IBITS
  組み込み関数 598
  特定名 599
IBM 自由ソース形式 21
IBSET
  組み込み関数 599
  特定名 599
ICHAR
  組み込み関数 600
  特定名 600
ID 指定子
  READ ステートメントの 446
  WAIT ステートメントの 496
  WRITE ステートメントの 501
idate_ サービスおよびユーティリティー・サブプログラム 893
IDIM 特定名 581
IDINT 特定名 604
IDNINT 特定名 640
IEEE 演算子 853
IEEE プロシージャー 854
IEEE モジュールとサポート 849
IEEE_CLASS 855
IEEE_CLASS_TYPE 852

IEEE_COPY_SIGN　856
IEEE_FEATURES_TYPE　853
IEEE_FLAG_TYPE　851
IEEE_GET_FLAG　856
IEEE_GET_HALTING　857
IEEE_GET_ROUNDING　857
IEEE_GET_STATUS　858
IEEE_IS_FINITE　858
IEEE_IS_NAN　859
IEEE_IS_NEGATIVE　859
IEEE_IS_NORMAL　860
IEEE_LOGB　861
IEEE_NEXT_AFTER　861
IEEE_REM　862
IEEE_RINT　863
IEEE_ROUND_TYPE　853
IEEE_SCALB　863
IEEE_SELECTED_REAL_KIND　864
IEEE_SET_FLAG　865
IEEE_SET_HALTING　865
IEEE_SET_ROUNDING　866
IEEE_SET_STATUS　866
IEEE_STATUS_TYPE　852
IEEE_SUPPORT_DATATYPE　867
IEEE_SUPPORT_DENORMAL　867
IEEE_SUPPORT_DIVIDE　868
IEEE_SUPPORT_FLAG　868
IEEE_SUPPORT_HALTING　869
IEEE_SUPPORT_INF　869
IEEE_SUPPORT_IO　869
IEEE_SUPPORT_NAN　870
IEEE_SUPPORT_ROUNDING　870
IEEE_SUPPORT_SQRT　871
IEEE_SUPPORT_STANDARD　872
IEEE_UNORDERED　872
IEEE_VALUE　873
IEOR
　組み込み関数　600
　特定名　601
ierrno_ サービスおよびユーティリティー・サブプログラム　893
IF
　構造体　147
　ステートメント
　　算術　386
　　ブロック　387
IF *(続き)*
　ステートメント *(続き)*
　　論理　388
IFIX 特定名　604
ILEN 組み込み関数　601
IMAG
　組み込み関数　602
　初期化式　110
IMPLICIT
　ステートメント、ストレージ・クラスの割り当て　79
　説明　389
　タイプの決め方　69
INCLUDE コンパイラー・ディレクティブ　517
INDEPENDENT コンパイラー・ディレクティブ　519
INDEX
　組み込み関数　602
　初期化式　110
　特定名　603
INQUIRE ステートメント　392
INT
　組み込み関数　603
　初期化式　110
　特定名　604
INT2 組み込み関数　605
INTEGER タイプ宣言ステートメント　399
INTENT 属性　405
IOR
　組み込み関数　606
　特定名　606
IOSTAT 値　229
IOSTAT 指定子
　BACKSPACE ステートメントの　296
　CLOSE ステートメントの　313
　ENDFILE ステートメントの　357
　INQUIRE ステートメントの　392
　OPEN ステートメントの　421
　READ ステートメントの　446
　REWIND ステートメントの　462
　WAIT ステートメントの　496
　WRITE ステートメントの　501
IQINT 特定名　604
IQNINT 特定名　640
irand サービスおよびユーティリティー・サブプログラム　894
irtc サービスおよびユーティリティー・サブプログラム　894
ISHFT
　組み込み関数　607
　特定名　607
ISHFTC
　組み込み関数　608
　特定名　608
ISIGN 特定名　667
ISYNC コンパイラー・ディレクティブ　906
itime_ サービスおよびユーティリティー・サブプログラム　895
I/O 条件　229
I/O リストと形式仕様の相互作用　269

J

jdate サービスおよびユーティリティー・サブプログラム　895

K

KIND
　組み込み、定数式　109
　組み込み関数　609
　組み込み制限式　111
kind タイプ・パラメーター　27

L

L (論理) 編集　256
LANGLVL 実行時オプション　237, 277
LBOUND 配列組み込み関数　609
LEADZ 組み込み関数　610
LEN
　組み込み、定数式　109
　組み込み関数　611
　組み込み制限式　111
　特定名　611

lenchr_ サービスおよびユーティリティー・サブプログラム　895
length 型付きパラメーター　27
LEN_TRIM 組み込み関数　612
LGAMMA
　組み込み関数　612
　特定名　613
LGE
　組み込み関数　613
　特定名　614
LGT
　組み込み関数　614
　特定名　615
LIGHT_SYNC コンパイラー・ディレクティブ　906
LLE
　組み込み関数　615
　特定名　616
LLT
　組み込み関数　616
　特定名　617
lnblnk_ サービスおよびユーティリティー・サブプログラム　896
LOC
　組み込み関数　144, 617
LOG 組み込み関数　618
LOG10 組み込み関数　619
LSHIFT
　エレメント型関数　620
　特定名　621
ltime_ サービスおよびユーティリティー・サブプログラム　896

M

MASTER コンパイラー・ディレクティブ　711
MATMUL 配列組み込み関数　621
MAX
　組み込み関数　623
　初期化式　110
MAX0 特定名　624
MAX1 特定名　624
MAXEXPONENT 組み込み関数　625
MAXLOC 配列組み込み関数　625
MAXVAL 配列組み込み関数　627
mclock サービスおよびユーティリティー・サブプログラム　897
MERGE 配列組み込み関数　629
MIN
　組み込み関数　630
　初期化式　110
MIN0 特定名　631
MIN1 特定名　631
MINEXPONENT 組み込み関数　631
MINLOC 配列組み込み関数　632
MINVAL 配列組み込み関数　634
MOD
　組み込み関数　635
　初期化式　110
　特定名　636
MODULE PROCEDURE ステートメント　417
MODULO 組み込み関数　636
MTSF PowerPC 組み込み関数　916
MTSFI PowerPC 組み込み関数　917
MULHY PowerPC 組み込み関数　917
MVBITS 組み込みサブルーチン　637

N

name
　共通ブロック　315
NAME 指定子、INQUIRE ステートメントの　392
NAMED 指定子、INQUIRE ステートメントの　392
NAMELIST
　実行時オプション　281
　ステートメント　418
NEAREST 組み込み関数　638
NEQV 論理演算子　120
NEXTREC 指定子
　INQUIRE ステートメントの　392
NINT
　組み込み関数　639
　初期化式　110
　特定名　640
NML 指定子
　READ ステートメントの　446
　WRITE ステートメントの　501
NOT
　組み込み関数　640
　特定名　640
　論理演算子　120
NULL
　組み込み関数　641
　初期化式　110
NULLIFY ステートメント　419
NUM 指定子
　READ ステートメントの　446
　WRITE ステートメントの　501
NUMBER 指定子、INQUIRE ステートメントの　392
NUMBER_OF_PROCESSORS 組み込み関数　643
NUM_PARTHDS 照会組み込み関数　642
NUM_USRTHDS 照会組み込み関数　644

O

O (8 進) 編集　257
omp_destory_nest_lock OpenMP ネスト可能ロック・ルーチン　775
omp_destroy_lock OpenMP ロック・ルーチン　774
omp_get_dynamic 実行環境ルーチン　775
omp_get_max_threads 実行環境ルーチン　775
omp_get_nested 実行環境ルーチン　776
omp_get_num_procs 実行環境ルーチン　776
omp_get_num_threads 実行環境ルーチン　777
omp_get_thread_num 実行環境ルーチン　777
omp_get_wtick OpenMP タイミング・ルーチン　778
omp_get_wtime OpenMP タイミング・ルーチン　779
omp_init_lock ロック・ルーチン　780

omp_init_nest_lock OpenMP ネスト可能ロック・ルーチン 781
omp_in_parallel 実行環境ルーチン 779
omp_set_dynamic 実行環境ルーチン 781
omp_set_lock ロック・ルーチン 782
omp_set_nested 実行環境ルーチン 783
omp_set_nest_lock ネスト可能ロック・ルーチン 783
omp_set_num_threads 実行環境ルーチン 784
omp_test_lock ロック・ルーチン 784
omp_test_nest_lock ロック・ルーチン 785
omp_unset_lock ロック・ルーチン 786
omp_unset_nest_lock ロック・ルーチン 786
OPEN ステートメント 420
OPENED 指定子、INQUIRE ステートメントの 392
OpenMP
 実行環境ルーチン
  説明 773
  omp_get_dynamic 775
  omp_get_max_threads 775
  omp_get_nested 776
  omp_get_num_procs 776
  omp_get_num_threads 777
  omp_get_thread_num 777
  omp_in_parallel 779
  omp_set_dynamic 781
  omp_set_nested 783
  omp_set_num_threads 784
 タイミング・ルーチン
  omp_get_wtick 778
  omp_get_wtime 779
 ネスト可能ロック・ルーチン
  omp_destroy_nest_lock 775
  omp_init_nest_lock 781
  omp_set_nest_lock 783
 ロック・ルーチン
  説明 773
OpenMP *(続き)*
 ロック・ルーチン *(続き)*
  omp_destroy_lock 774
  omp_init_lock 780
  omp_set_lock 782
  omp_test_lock 784
  omp_test_nest_lock 785
  omp_unset_lock 786
  omp_unset_nest_lock 786
OPTIONAL 属性 427
OR
 特定名 606
 論理演算子 120
ORDERED コンパイラー・ディレクティブ 712

P

P (スケール因数) 編集 266
PACK 配列組み込み関数 644
PAD 指定子
 INQUIRE ステートメントの 392
 OPEN ステートメントの 421
PARALLEL DO コンパイラー・ディレクティブ
 説明 718
 SCHEDULE 文節 718
PARALLEL SECTIONS コンパイラー・ディレクティブ
 説明 722
PARALLEL WORKSHARE コンパイラー・ディレクティブ
 説明 726
PARALLEL コンパイラー・ディレクティブ
 説明 715
PARAMETER 属性 429
PAUSE ステートメント 430
PERMUTATION コンパイラー・ディレクティブ 525
pointee
 POINTER ステートメントおよび 434
POSITION 指定子
 INQUIRE ステートメントの 392
 OPEN ステートメントの 421
PRECISION 組み込み関数 646
PREFETCH_BY_LOAD コンパイラー・ディレクティブ 906
PREFETCH_BY_STREAM_BACKWARD コンパイラー・ディレクティブ 906
PREFETCH_BY_STREAM_FORWARD コンパイラー・ディレクティブ 906
PREFETCH_FOR_LOAD コンパイラー・ディレクティブ 906
PREFETCH_FOR_STORE コンパイラー・ディレクティブ 906
PRESENT 組み込み関数 427, 647
PRINT ステートメント 436
PRIVATE
 ステートメント 42, 438
 属性 438
PROCESSORS_SHAPE 組み込み関数 648
PRODUCT 配列組み込み関数 648
PROGRAM ステートメント 441
PROTECTED 属性 442
Pthreads ライブラリー、AIX 787
Pthreads ライブラリー・モジュール
 関数の説明 787
 f_maketime 関数 790
 f_pthread_attr_destroy 関数 790
 f_pthread_attr_getdetachstate 関数 791
 f_pthread_attr_getguardsize 関数 791
 f_pthread_attr_getinheritsched 関数 792
 f_pthread_attr_getschedparam 関数 792
 f_pthread_attr_getschedpolicy 関数 793
 f_pthread_attr_getscope 関数 793
 f_pthread_attr_getstackaddr 794
 f_pthread_attr_getstacksize 関数 795
 f_pthread_attr_init 関数 795
 f_pthread_attr_setdetachstate 関数 796

Pthreads ライブラリー・モジュール *(続き)*
f_pthread_attr_setguardsize 関数 796
f_pthread_attr_setinheritsched 関数 797
f_pthread_attr_setschedparam 関数 798
f_pthread_attr_setschedpolicy 関数 798
f_pthread_attr_setscope 関数 799
f_pthread_attr_setstackaddr 関数 800
f_pthread_attr_setstacksize 関数 800
f_pthread_attr_t 関数 801
f_pthread_cancel 関数 801
f_pthread_cleanup_pop 関数 801
f_pthread_cleanup_push 関数 802
f_pthread_condattr_destroy 関数 807
f_pthread_condattr_getpshared 関数 807
f_pthread_condattr_init 関数 808
f_pthread_condattr_setpshared 関数 808
f_pthread_condattr_t 関数 809
f_pthread_cond_broadcast 関数 803
f_pthread_cond_destroy 関数 804
f_pthread_cond_init 関数 804
f_pthread_cond_signal 関数 805
f_pthread_cond_t 関数 805
f_pthread_cond_timedwait 関数 805
f_pthread_cond_wait 関数 806
f_pthread_create 関数 809
f_pthread_detach 関数 810
f_pthread_equal 関数 811
f_pthread_exit 関数 811
f_pthread_getconcurrency 関数 812
f_pthread_getschedparam 関数 812
f_pthread_getspecific 関数 813
f_pthread_join 関数 813
f_pthread_key_create 関数 814

Pthreads ライブラリー・モジュール *(続き)*
f_pthread_key_delete 関数 814
f_pthread_key_t 815
f_pthread_kill 関数 815
f_pthread_mutexattr_destroy 関数 819
f_pthread_mutexattr_getprioceiling 関数 820
f_pthread_mutexattr_getprotocpol 関数 820
f_pthread_mutexattr_getpshared 関数 820
f_pthread_mutexattr_gettype 関数 821
f_pthread_mutexattr_init 関数 822
f_pthread_mutexattr_setprioceiling 関数 822
f_pthread_mutexattr_setprotocol 関数 823
f_pthread_mutexattr_setpshared 関数 823
f_pthread_mutexattr_settype 関数 824
f_pthread_mutexattr_t 825
f_pthread_mutex_destroy 関数 816
f_pthread_mutex_getprioceiling 関数 816
f_pthread_mutex_init 関数 816
f_pthread_mutex_lock 関数 817
f_pthread_mutex_setprioceiling 関数 818
f_pthread_mutex_t 818
f_pthread_mutex_trylock 関数 818
f_pthread_mutex_unlock 関数 819
f_pthread_once 関数 825
f_pthread_once_t 826
f_pthread_rwlockattr_destroy 関数 831
f_pthread_rwlockattr_getpshared 関数 831
f_pthread_rwlockattr_init 関数 832
f_pthread_rwlockattr_setpshared 関数 832
f_pthread_rwlockattr_t 関数 833

Pthreads ライブラリー・モジュール *(続き)*
f_pthread_rwlock_destroy 関数 826
f_pthread_rwlock_init 関数 826
f_pthread_rwlock_rdlock 関数 827
f_pthread_rwlock_t 関数 828
f_pthread_rwlock_tryrdlock 関数 828
f_pthread_rwlock_trywrlock 関数 829
f_pthread_rwlock_unlock 関数 830
f_pthread_rwlock_wrlock 関数 830
f_pthread_self 関数 833
f_pthread_setcancelstate 関数 833
f_pthread_setcanceltype 関数 834
f_pthread_setchedparam 関数 835
f_pthread_setconcurrency 関数 834
f_pthread_setspecific 関数 836
f_pthread_t 関数 836
f_pthread_testcancel 関数 837
f_sched_param 関数 837
f_sched_yield 関数 837
f_timespec 関数 837
PUBLIC 属性 444
PURE 209

Q

Q (拡張精度) 編集 247
QABS 特定名 547
QACOS 特定名 548
QACOSD 特定名 549
QARCOS 特定名 548
QARSIN 特定名 556
QASIN 特定名 556
QASIND 特定名 557
QATAN 特定名 559
QATAN2 特定名 561
QATAN2D 特定名 562
QATAND 特定名 559
QCMPLX
組み込み関数 650
初期化式 110

QCMPLX *(続き)*
特定名 651
QCONJG 特定名 568
QCOS 特定名 568
QCOSD 特定名 569
QCOSH 特定名 570
QDIM 特定名 581
QERF 特定名 586
QERFC 特定名 587
QEXP 特定名 588
QEXT
組み込み関数 651
初期化式 110
特定名 652
QEXTD 特定名 652
QFLOAT 特定名 652
QGAMMA 特定名 592
QIMAG 特定名 551
QINT 特定名 552
QLGAMA 特定名 613
QLOG 特定名 618
QLOG10 特定名 619
QMAX1 特定名 624
QMIN1 特定名 631
QMOD 特定名 636
QNINT 特定名 554
QPROD 特定名 583
QREAL 特定名 658
QSIGN 特定名 667
QSIN 特定名 669
QSIND 特定名 670
QSINH 特定名 671
qsort_ サービスおよびユーティリティー・サブプログラム 897
qsort_down サービスおよびユーティリティー・サブプログラム 898
qsort_up サービスおよびユーティリティー・サブプログラム 898
QSQRT 特定名 677
QTAN 特定名 683
QTAND 特定名 684
QTANH 特定名 684

R

RADIX 組み込み関数 652
RAND 組み込み関数 652
RANDOM_NUMBER 組み込みサブルーチン 653
RANDOM_SEED 組み込みサブルーチン 654
RANGE 組み込み関数 656
READ
指定子、INQUIRE ステートメントの 392
ステートメント 445
READWRITE 指定子、INQUIRE ステートメントの 392
REAL
組み込み関数 657
初期化式 110
特定名 658
REAL タイプ宣言ステートメント 453
REC 指定子
READ ステートメントの 446
WRITE ステートメントの 501
RECL 指定子
INQUIRE ステートメントの 392
OPEN ステートメントの 421
RECORD ステートメント 458
RECURSIVE キーワード 380, 473
REPEAT
組み込み関数 111, 658
組み込み初期化式 110
RESHAPE
配列組み込み関数 111, 658
配列組み込み初期化式 110
RESULT キーワード 360, 379
RETURN ステートメント 460
REWIND ステートメント 461
ROTAELI PowerPC 組み込み関数 917
ROTAELM PowerPC 組み込み関数 918
RRSPACING 組み込み関数 660
RSHIFT
エレメント型関数 660
特定名 661
rtc サービスおよびユーティリティー・サブプログラム 899

S

S (符号制御) 編集 267
SAVE 属性 463
SCALE 組み込み関数 661
SCAN
組み込み関数 662
初期化式 110
SCHEDULE コンパイラー・ディレクティブ 726
説明 726
SCHEDULE 文節
DO (作業共用) ディレクティブの 703
SCHEDULE 文節、PARALLEL DO ディレクティブの 718
SECTIONS コンパイラー・ディレクティブ
説明 730
section_subscript、配列セクションの構文 95
SELECT CASE ステートメント
説明 465
CASE 構造体 149
CASE ステートメントおよび 304
SELECTED_INT_KIND
組み込み関数 111, 663
組み込み初期化式 110
SELECTED_REAL_KIND
組み込み関数 111, 664
組み込み初期化式 110
SEQUENCE ステートメント 42, 467
SEQUENTIAL 指定子、INQUIRE ステートメントの 392
SETFSB0 PowerPC 組み込み関数 918
SETFSB1 PowerPC 組み込み関数 918
setrteopts サービスおよびユーティリティー・サブプログラム 899
SET_EXPONENT 組み込み関数 665

set_fpscr サブプログラム 848
set_fpscr_flags サブプログラム 848
set_round_mode サブプログラム 849
SFTI PowerPC 組み込み関数 919
SIGN
組み込み関数 666
初期化式 110
特定名 667
SIGNAL 組み込みサブルーチン 668
signal.h インクルード・ファイル 668
SIN
組み込み関数 669
特定名 669
SIND
組み込み関数 670
特定名 670
SINGLE / END SINGLE コンパイラー・ディレクティブ 734
SINH
組み込み関数 671
特定名 671
SIZE
指定子、READ ステートメントの 446
配列組み込み関数 671
SIZEOF
組み込み関数 672
sleep_ サービスおよびユーティリティー・サブプログラム 900
SMP
概要 695
ディレクティブ 695
SNAPSHOT コンパイラー・ディレクティブ 527
SNGL 特定名 658
SNGLQ 特定名 658
SOURCEFORM コンパイラー・ディレクティブ 529
SP (符号制御) 編集 267
SPACING 組み込み関数 674
specification_part 182
SPREAD 配列組み込み関数 675
SQRT
組み込み関数 676
特定名 677
SRAND 組み込みサブルーチン 677
SS (符号制御) 編集 267
STATIC
属性 470
STATUS 指定子
CLOSE ステートメントの 313
OPEN ステートメントの 421
STOP ステートメント 472
STREAM_UNROLL コンパイラー・ディレクティブ 530
SUBSCRIPTORDER コンパイラー・ディレクティブ 532
subscript_triplet の構文 97
SUM 配列組み込み関数 678
SYSTEM 組み込みサブルーチン 680
SYSTEM_CLOCK 組み込みサブルーチン 681

T

T (定位置) 編集 268
TAN
組み込み関数 682
特定名 683
TAND
組み込み関数 683
特定名 684
TANH
組み込み関数 684
特定名 684
TARGET 属性 475
THREADLOCAL コンパイラー・ディレクティブ 738
THREADPRIVATE コンパイラー・ディレクティブ 741
timef サービスおよびユーティリティー・サブプログラム 900
timef_delta サービスおよびユーティリティー・サブプログラム 901
time_ サービスおよびユーティリティー・サブプログラム 900
TINY 組み込み関数 685
TL (定位置) 編集 268
TR (定位置) 編集 268
TRANSFER 組み込み関数
初期化式 110
制限式 111
説明 685
TRANSFER 指定子、INQUIRE ステートメントの 392
TRANSPOSE 配列組み込み関数 687
TRAP PowerPC 組み込み関数 919
TRIM 組み込み関数
初期化式 110
制限式 111
説明 687
TZ 環境変数 576

U

UBOUND 配列組み込み関数 688
umask_ サービスおよびユーティリティー・サブプログラム 901
UNFORMATTED 指定子
INQUIRE ステートメントの 392
Unicode 文字およびファイル名
環境変数 39
コンパイラー・オプション 39
ホレリス定数 66
文字定数 39, 264
H 編集および 265
UNIT 指定子
BACKSPACE ステートメントの 296
CLOSE ステートメントの 313
ENDFILE ステートメントの 357
INQUIRE ステートメントの 392
OPEN ステートメントの 421
READ ステートメントの 446
REWIND ステートメントの 462
WRITE ステートメントの 501
UNPACK 配列組み込み関数 689
UNROLL コンパイラー・ディレクティブ 534
UNROLL_AND_FUSE コンパイラー・ディレクティブ 536
USE ステートメント 488
USE ステートメントの ONLY 文節 489

usleep_ サービスおよびユーティリティー・サブプログラム 902

V

VALUE 属性 491
VERIFY
組み込み関数 690
初期化式 110
VIRTUAL ステートメント 493
VOLATILE 属性 493

W

WAIT ステートメント 496
WHERE
構造体 131
構造体ステートメント 497
ステートメント 131, 497
FORALL でネストされた 140
where_construct_name 131, 348, 351, 497
WORKSHARE コンパイラー・ディレクティブ 747
WRITE
ステートメント 500
INQUIRE ステートメントの指定子 392

X

X (定位置) 編集 268
xlfutility モジュール 879
xlf_fp_util モジュール 842
xl_ _trbk サービスおよびユーティリティー・サブプログラム 902
XOR
特定名 601
論理演算子 120

Z

Z (16 進) 編集 260
ZABS 特定名 547
ZCOS 特定名 568
ZEXP 特定名 588
ZLOG 特定名 618
ZSIN 特定名 669
ZSQRT 特定名 677

[特殊文字]

! インライン注釈 15, 16
″″ (二重引用符) 編集 263
#line コンパイラー・ディレクティブ 523
$ (ドル記号) 編集 263
’ (アポストロフィ) 編集 263
* 注釈行 16
+、-、*、/、** 算術演算子 114
/ (スラッシュ) 編集 262
// (連結) 演算子 116
: (コロン) 編集 262
:: (ダブル・コロン) セパレーター 299
; ステートメント・セパレーター 18, 20
%VAL および %REF 関数 197
@PROCESS コンパイラー・ディレクティブ 526
_OPENMP C プリプロセッサー・マクロ 23

プログラム番号: 5765-F70

SC88-9395-01

日本アイ・ビー・エム株式会社
〒106-8711　東京都港区六本木3-2-12