IBM C for VSE/ESA

Language Reference

Release 1

SC09-2425-00

IBM C for VSE/ESA

Language Reference

Release 1

SC09-2425-00

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

First Edition (December 1996)

This edition applies to Version 1, Release 1, Modification Level 0, of IBM C for VSE/ESA (Program 5686-A01); Version 1, Release 4, Modification Level 0, of IBM Language Environment for VSE/ESA (Program 5686-094); the VSE C Language Run-Time Support feature of VSE/ESA Version 2 Release 2 (Program 5690-VSE); and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers’ comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. See “Communicating Your Comments to IBM” for a description of the methods. This page immediately precedes the Readers’ Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

Programming Interface Information

This book is intended to help the customer program with the IBM C for VSE/ESA language. This book documents General-Use Programming Interfaces and associated guidance information provided by the IBM C for VSE/ESA and IBM Language Environment for VSE/ESA products.

General-Use Programming Interfaces allow the customer to write programs that obtain the services of the IBM C for VSE/ESA compiler and IBM Language Environment for VSE/ESA.

Standards

Extracts are reprinted from *IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language]*, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Trademarks

About This Book

This book provides you with a description of the IBM C for VSE/ESA (C/VSE) language definition as implemented for the IBM Language Environment for VSE/ESA (LE/VSE) environment. It is intended for use by programmers who need to understand the support provided by the C/VSE compiler.

Note: References to LE/VSE also apply to the VSE C Language Run-Time Support feature of VSE/ESA Version 2 Release 2.

Implementation-defined behavior is any action that is not defined by the standards but by the implementing compiler and library. Refer to the *LE/VSE C Run-Time Programming Guide* for information about implementation-defined behavior in the LE/VSE environment.

Undefined behavior is any action, by the compiler and library on an erroneous program, that does not result in any expected manner. You should not write any programs to rely on such behavior.

Unspecified behavior is any other action by the compiler and library that is not defined by the standards.

The C Language

The C language is a general purpose, function-oriented programming language that allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages, and it also provides many of the benefits of a low-level language. Using the C/VSE language, you can write portable code conforming to the ANSI standard.

IBM offers the C language on other platforms, such as the OS/2, AIX/6000, OS/400, OS/390, and VM operating systems.

The elements of the C/VSE implementation include:

- All elements of the joint ISO and IEC standard: ISO/IEC 9899:1990 (E)
- ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
- Locale based internationalization support as defined in: ISO/IEC DIS 9945-2:1992/IEEE POSIX 1003.2-1992 Draft 12 (There are some limitations to fully-compliant behavior as noted in the *LE/VSE C Run-Time Programming Guide*.)
- Extended multibyte and wide character utilities as defined by a subset of the Programming Language C Amendment 1, which will be ISO/IEC 9899:1990/Amendment 1:1994(E)

IBM Language Environment for VSE/ESA

C/VSE exploits the C run-time environment and library of run-time callable services provided by IBM Language Environment for VSE/ESA (LE/VSE).

LE/VSE establishes a common run-time environment and common run-time callable services for language products, user programs, and other products.

The common execution environment is made up of data items and services performed by library routines available to a particular application running in the environment. The services that LE/VSE provides to your application may include:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, support for interlanguage communication (ILC), and condition handling.
- Extended services often needed by applications. These functions are contained within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Run-time options that help the execution, performance tuning, performance, and diagnosis of your application.
- Access to language-specific library routines, such as the C functions.

Using Your Documentation

The publications in the C/VSE and LE/VSE libraries are designed to help you develop C/VSE applications that run with LE/VSE. Each publication helps you perform a different task. For a complete list of publications you might need, see “Bibliography” on page 189. Table 1 lists the publications in the C/VSE library.

Table 1. How to Use C/VSE Publications

To...	Use...	
Plan for, install, customize, and maintain C/VSE	*Installation and Customization Guide*	GC09-2422
Migrate VSE applications from C/370 to C/VSE	*Migration Guide*	SC09-2423
Get details on C/VSE syntax and specifications of language elements	*Language Reference*	SC09-2425
Find syntax for compile-time options; compile your C/VSE applications; get details on compile-time messages	*User's Guide*	SC09-2424
Diagnose compiler problems and report them to IBM	*Diagnosis Guide*	GC09-2426
Understand warranty information	*Licensed Program Specifications*	GC09-2421

Table 2 on page xi lists the publications in the LE/VSE library. These include publications designed to help you develop and debug your C/VSE applications, diagnose run-time problems that occur in your C/VSE applications, and use C/VSE-related utilities.

Table 2. How to Use LE/VSE Publications

To...	Use...	
Evaluate LE/VSE	*Fact Sheet* *Concepts Guide*	GC33-6679 GC33-6680
Plan for, install, customize, and maintain LE/VSE	*Installation and Customization Guide*	SC33-6682
Understand the LE/VSE program models and concepts	*Concepts Guide* *Programming Guide*	GC33-6680 SC33-6684
Find syntax for LE/VSE run-time options and callable services	*Programming Reference*	SC33-6685
Develop your C/VSE applications	*Programming Guide* *C Run-Time Programming Guide* *C Run-Time Library Reference*	SC33-6684 SC33-6688 SC33-6689
Develop interlanguage communication (ILC) applications	*Writing Interlanguage Communication Applications*	SC33-6686
Debug your C/VSE applications and get details on run-time messages	*Debugging Guide and Run-Time Messages*	SC33-6681
Migrate applications to LE/VSE	*Run-Time Migration Guide*	SC33-6687
Diagnose run-time problems that occur in your C/VSE applications	*Debugging Guide and Run-Time Messages*	SC33-6681
Use C/VSE-related utilities	*C Run-Time Programming Guide*	SC33-6688
Understand warranty information	*Licensed Program Specifications*	GC33-6683

Softcopy Examples

Most examples in the following books are available in machine-readable form:

- *C/VSE Installation and Customization Guide*, GC09-2422
- *C/VSE User's Guide*, SC09-2424
- *C/VSE Language Reference*, SC09-2425

Softcopy examples are indicated in the book by a label in the form, `EDCX`*bnnn*. The *b* refers to the book:

- `I` is the *C/VSE Installation and Customization Guide*
- `U` is the *C/VSE User's Guide*
- `R` is the *C/VSE Language Reference*

Softcopy examples are installed on your system along with C/VSE, in the sublibrary `PRD2.DBASE`.

Example member names are the same as the labels indicated in the book.

Contact your system programmer if the default names are not used at your installation.

How to Read the Syntax Diagrams

In this book, syntax for commands, directives, and statements is described using the following structure:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  A double right-arrowhead indicates the beginning of a command, directive, or statement; the single right-arrowhead indicates that it is continued on the next line. (In the following diagrams, `statement` is used to represent a command, directive, or statement.)

  ```
  ►►──statement──►
  ```

  The following indicates a continuation; the opposing arrowheads indicate the end of a command, directive, or statement.

  ```
  ►──statement──►◄
  ```

  Diagrams of syntactical units other than complete commands, directives, or statements look like this:

  ```
  ►──statement──►
  ```

- Required items are on the horizontal line (the main path).

  ```
  ►►──statement──required_item──►◄
  ```

- IBM-supplied default items are above the main path.

  ```
                ┌─default_item─┐
  ►►──statement──┴──────────────┴──►◄
  ```

- Optional items are below the main path.

  ```
  ►►──statement──┬───────────────┬──►◄
                 └─optional_item─┘
  ```

- If you can choose from two or more items, they are vertical in a stack.

  If you *must* choose one of the items, one item of the stack is on the main path.

  ```
  ►►──statement──┬─required_choice1─┬──►◄
                 └─required_choice2─┘
  ```

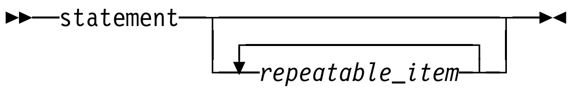
  If choosing one of the items is optional, the entire stack is below the main path.

  ```
  ►►──statement──┬──────────────────┬──►◄
                 ├─optional_choice1─┤
                 └─optional_choice2─┘
  ```

- An arrow returning to the left above a line indicates an item that you can repeat.

  ```
                 ┌────────────────┐
  ►►──statement──▼─repeatable_item─┴──►◄
  ```

  or

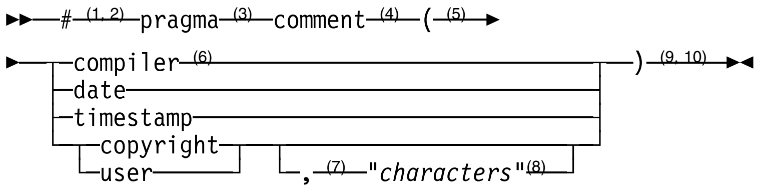


A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- Keywords are in non-italic letters and should be entered exactly as shown (for example, `pragma`). They must be spelled exactly as shown. Variables are in italics and lowercase letters (for example, *identifier*). They represent user-supplied names or values.
- Keywords that appear in mixed-case letters (for example, `AGGregate`) indicate that the keyword can be abbreviated (`AGG`) or entered in full (`AGGREGATE`).
- If punctuation marks, parentheses, arithmetic operators, or other non-alphanumeric characters are shown, you must enter them as part of the syntax.

Note: The white space is not always required between tokens but you should include at least one blank space between tokens unless otherwise specified.

The following syntax diagram example shows the syntax for the `#pragma comment` directive.



Notes:

1 This is the start of the syntax diagram.

2 The symbol `#` must appear first.

3 The keyword `pragma` must follow the `#` symbol.

4 The keyword `comment` must follow the keyword `pragma`.

5 An opening parenthesis must follow the keyword `comment`.

6 The comment type must be entered only as one of the following: `compiler`, `date`, `timestamp`, `copyright`, or `user`.

7 If the comment type is `copyright` or `user`, and an optional character string is following, a comma must be present after the comment type.

8 A character string must follow the comma. The character string must be enclosed in double quotation marks.

9 A closing parenthesis is required.

10 This is the end of the syntax diagram.

The following examples of the `#pragma comment` directive are syntactically correct according to the diagram shown above:

```
#pragma comment(date)
#pragma comment(user)
#pragma comment(copyright,"This text will appear in the module")
```

Chapter 1. Introduction to C

This chapter introduces you to the C programming language and shows you how to structure C language source programs. Refer to the *LE/VSE C Run-Time Programming Guide* for information about implementation-defined behavior in the C/VSE environment.

Overview of the C Language

C is a programming language designed for a wide variety of programming tasks. It is used for system-level code, text processing, graphics, and in many other application areas.

The C language contains a concise set of statements, with functionality added through its library. This division enables C to be both flexible and efficient. An additional benefit is that the language is highly consistent across different operating systems.

The C library contains functions for input and output, mathematics, exception handling, string and character manipulation, dynamic memory management, as well as date and time manipulation. Use of this library helps to maintain program portability, because the underlying implementation details for the various operations need not concern the programmer. Details about the C library can be found in the *LE/VSE C Run-Time Library Reference*.

C supports numerous data types, including characters, integers, floating-point numbers and pointers—each in a variety of forms. In addition, C also supports arrays, structures (records), unions, and enumerations.

C Source Programs

A *C source program* is a collection of one or more directives, declarations, and statements contained in one or more source files.

Statements	Specify the action to be performed.
Directives	Instruct the C preprocessor to act on the text of the program.
Declarations	Establish names and define characteristics such as scope, data type and linkage.
Definitions	Are declarations that allocate storage for data objects or define a body for functions. An object definition allocates storage and may optionally initialize the object.

A function definition contains the function body. The *function body* is a compound statement that can contain declarations and statements that define what the function does. The function definition declares the function name, its parameters, and the data type of the value it returns.

The order and scope of declarations affect how you can use variables and functions in statements. In particular, an identifier can be used only after it is declared.

A program must contain at least one function definition. If the program contains only one function definition, the function must be called `main()`. If the program contains more than one function definition, only one of the functions can be called `main()`. The `main()` function is the first function called when a program is run.

This is the source code of a simple C program:

EDCXRAAA

```
 /* Example of a simple C program */
#include <stdio.h>              /* standard library header that
                                   contains I/O function declarations
                                   such as printf used below         */

#include <math.h>               /* standard library header that
                                   contains math function declarations
                                   such as cos used below            */

#define NUM 46.0                /* Preprocessor directive            */

double x = 45.0;                /* Global variable
                                   definitions                       */

double y = NUM;

int main(void)                  /* Function definition
                                   for main function                 */
{
   double z;                    /* Internal variable                 */
   double w;                    /* definitions                       */

   z = cos(x);                  /* cos is declared in math.h as
                                   double cos(double arg)            */
   w = cos(y);
   printf ("cosine of x is %f\n", z);  /* printf is declared in      */
   printf ("cosine of y is %f\n", w);  /* stdio.h as                 */
                                    /* int printf (const char *, ...) */
}
```

This source program defines `main()`, the global variables `x` and `y`, and the local variables `z` and `w`. The source program declares a reference to the functions `cos()` and `printf()`.

C Source Files

A *C source file* is a text file that contains all or part of your C program. It can include any of the functions that the program needs. To create an executable object module, you compile the separate source files individually and then link them as one program. With the `#include` directive, you can combine source files into larger source files. The resulting collection of files constitutes a compilation unit.

A source file contains any combination of directives, declarations, and definitions. You can split items such as function definitions and large data structures between text files, but you cannot split them between object files. Before the source file is compiled, the preprocessor filters out preprocessor directives that may change the files. As a result of the preprocessing stage, preprocessor directives are completed, macros are expanded, and a source file is created containing C statements, completed directives, declarations, and definitions.

Sometimes you may find it useful to place variable definitions in one source file and declare references to those variables in any source files that use them. This procedure makes definitions easy to find and change, if necessary. You can also organize constants and macros into separate files and include them into source files as required.

Directives in a source file apply to that source file and its included files only. Each directive applies only to the part of the file following the directive.

The following example is a C program in two source files. The `main()` and `max()` functions are in separate files. The execution of the program begins with the `main()` function.

EDCXRAAB

```
 /* Example of a C program in two source files
    File 1 of 2 - file 2 is EDCXRMAX */

#include <stdio.h>

#define ONE     1
#define TWO     2
#define THREE   3

extern int max(int, int);      /* Function declaration */

int main(int argc, char * argv[])  /* Function definition */
{
   int u, w, x, y, z;

   u = 5;
   z = 2;
   w = max(u, ONE);
   x = max(w,TWO);
   y = max(x,THREE);
   z = max(y,z);

   printf("w is %d, x is %d, y is %d, z is %d",w,x,y,z);
}
```

EDCXRMAX

```
 /* Example of a C program in two source files
    File 2 of 2 - file 1 is EDCXRAAB */

int max (int a,int b)                /* Function  definition */
{
   if ( a > b )
       return (a);
   else
       return (b);
}
```

The first source file declares the function `max()`, but does not define it. Function `max()` is referenced in source file 1 and defined in source file 2. Four statements in `main()` are *function calls* of `max()`.

The lines beginning with a number sign (`#`) are preprocessor directives that direct the preprocessor to replace the identifiers `ONE`, `TWO`, and `THREE` with the numbers `1`, `2`, and `3`. The directives do not apply to the second source file.

The second source file contains the function definition for `max()`, which is called four times in `main()`. After you compile the source files, you can link and run them as a single program.

Program Execution

Every program must have a function called `main()`. Programs usually contain other functions as well.

The `main()` function is the starting point for running a program. The statements within the `main()` function are executed sequentially. They may be calls to other functions. A program usually stops running at the end of the `main()` function, although it can stop at other points in the program.

You can make your program more modular by creating separate functions to perform a specific task or set of tasks. The `main()` function calls these functions to perform the tasks. Whenever a function call is made, the statements are executed sequentially starting with the first statement in the function. The function returns control to the calling function at the `return` statement or at the end of the function.

You can declare any function to have parameters. When functions are called, they receive values for their parameters from the arguments passed by the calling functions. You can declare parameters for the `main()` function so you can pass values to `main()` from the command line. The command line that starts the program can pass such values as described in "main()."

main()

When you begin the execution of a program, the system automatically calls the function `main()`. Every program must have one function named `main()`, with the name `main` written in lowercase letters. A `main()` function has the form:

```
►►──┬─────┬──main──(──┬──────────────────────────────────┬──)──►
    └─int─┘           ├─void─────────────────────────────┤
                      └─int──argc──,──char──*──argv[]────┘
```

```
►──block_statement──►◄
```

The function `main()` can declare either no or two parameters. Although any name can be given to these parameters, they are usually referred to as *argc* and *argv*. The first parameter, *argc* (argument count), has type `int` and indicates how many arguments were entered on the command line when the program was invoked. The second parameter, *argv* (argument vector), has type array of pointers to `char` array objects. `char` objects are null-terminated strings.

The value of *argc* indicates the number of pointers in the array *argv*. If a program name is available, the first element in *argv* points to a character array that contains the program name or the invocation name of the program that is being executed. If the name cannot be determined, the first element in *argv* points to a null character. This name is counted as one of the arguments to the function `main()`. For example, if only the program name is entered on the command line, *argc* has a value of `1` and *argv[0]* points to the program name.

Regardless of the number of arguments entered on the command line, *argv[argc]* always contains `NULL`.

The following program `backward` prints the arguments entered on a command line such that the last argument is printed first:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s ", argv[argc]);
}
```

If you invoke this program from a command line with the following:

```
backward string1 string2
```

The output generated is:

```
string2 string1
```

The arguments *argc* and *argv* would contain the following values:

Object	**Value**
argc	3
argv[0]	pointer to string "backward"
argv[1]	pointer to string "string1"
argv[2]	pointer to string "string2"
argv[3]	NULL

Command-Line Arguments

Command-line arguments are treated differently in different environments:

Under Batch

`argc`	Returns the number of strings in the argument line
`argv[0]`	Returns the program name in uppercase
`argv[1 to` *n*`]`	Returns the arguments as they were entered

Under DL/I

`argc`	Returns 1
`argv[0]`	Is a null pointer

Under CICS

`argc`	Returns 1
`argv[0]`	Returns the transaction ID

You pass command-line arguments to a C/VSE application using the PARM parameter of the JCL EXEC statement or, if you are calling a C/VSE application from another C/VSE application, using the parameter string of the `system()` library function.

For more information on passing command-line arguments, see the *C/VSE User's Guide*.

Related Information

- “Calling Functions and Passing Arguments” on page 98
- “Function Declarator” on page 85
- “Type Specifiers” on page 44
- “Identifiers” on page 12
- “Block” on page 127

Chapter 2. Elements of C

This chapter describes the basic elements of the C programming language.

Character Set

This is the basic character set that must be available at both compile time and run time. It defines the source character set (characters used for writing the source code), and the execution character set (characters recognized at run time).

The C/VSE set supports the basic character set which maps onto the source and the execution character sets. This basic character set always consists of:

- The uppercase and lowercase letters of the English alphabet:

  ```
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  ```

- The decimal digits `0` through `9`:

  ```
  0 1 2 3 4 5 6 7 8 9
  ```

- The following graphic characters:

  ```
  ! " # % & ' ( ) * +
  , - . / : ; < = > ?
  [ \ ] _ { } ˜ ^ |
  ```

- The space character.
- The control characters representing horizontal tab, vertical tab, form feed, and end of string.

The execution character set also includes control characters representing alert, backspace, carriage return, and newline.

In a source file, a record contains one line of source text; the end of a record indicates the end of a source line.

The encoding of the following characters

```
! # ` [ ] \ { } ˜ ^ |
```

from the basic character set may vary between source character sets and between execution character sets. The C/VSE compiler normalizes the encoding of source files indicated by the `#pragma filetag` directive to the character set specified with the `LOCALE` option. Such normalized encoding is also used for execution character set encoding.

Depending on the EBCDIC encoding used in your installation, the two characters ^ and | may be expressed as ¬ and ¦, respectively. In this documentation, the ^ and

| symbols will be referred to as the circumflex and vertical bar, respectively. If the `NOLOCALE` option is specified, normalization is not performed, and character set encoding is assumed to be that of the IBM-1047 character set. In this case both the broken and unbroken vertical bars are recognized as the vertical bar, and the circumflex and logical not sign are recognized as the circumflex. For a detailed description of the `#pragma filetag` directive and the `LOCALE` option, refer to "Internationalization: Locales and Character Sets" in the *LE/VSE C Run-Time Programming Guide*.

The compiler recognizes and supports the additional characters (the extended character set) which can be meaningfully used in string literals and character constants. The support for extended characters includes the multibyte character sets. Also, any sequence of characters is allowed in comments.

Uppercase and lowercase letters are treated as distinct characters. If a lowercase `a` is specified as part of an identifier name, you cannot substitute an uppercase `A` in its place; you must use the lowercase letter.

Multibyte characters are represented on the System/370 system using Shiftout <SO> and Shiftin <SI> pairs. Strings are of the form:

```
<SO> x y z <SI>
```

or mixed

```
<SO> x <SI> y z
x <SO> y <SI> z
```

where each character between the <SO> and <SI> pairs is represented by two bytes. Multibyte characters are restricted to character constants, string constants, and comments. Refer to the *LE/VSE C Run-Time Library Reference* for a discussion on strings passed to library routines, and to "Character Constants" on page 26 of this book for information on character constants.

For the environments that do not support the entire character set, you can use trigraphs as alternative symbols to represent some characters.

Trigraphs

Some characters from the C character set are not available in all environments, or have an internal representation that is not consistent across EBCDIC systems. You can enter these characters into a C source program using a sequence of three characters called a *trigraph*.

At compile time, the compiler translates the trigraphs found in string literals and character constants into the decimal values of characters they represent in the coded character set selected by the `LOCALE` compile-time option.

Note: C/VSE will compile source files that were edited using different encoding of character sets. However, the compilation of source files that were edited with the following is not supported:

- A character set that does not support all the characters (or trigraphs) specified above
- A character set for which there is no one-to-one mapping between it and the character set above.

The trigraph sequences are:

`??=`	`#`
`??(`	`[`
`??)`	`]`
`??<`	`{`
`??>`	`}`
`??/`	`\`
`??'`	`^`
`??!`	`|`
`??-`	`~`

Note: The exclamation mark (!) is a variant character. Its recognition depends on whether or not the `LOCALE` option is active. For more information on variant characters, refer to the *LE/VSE C Run-Time Programming Guide*.

Example

```
some_array??(i??) = n;
```

represents

```
some_array[i] = n;
```

Escape Sequences

An escape sequence contains a backslash (\) symbol followed by either:

- One of the escape sequence characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `?`, `\`

or

- An octal or hexadecimal number. A hexadecimal escape sequence contains an `x` followed by one or more hexadecimal digits (0-9, A-F, a-f). An octal escape sequence contains one or more octal digits (0-7).

The value of the hexadecimal or octal number specifies the value of the desired character or wide character.

You can represent any member of the character set used at run time by an *escape sequence*. For example, you can use escape sequences to place such characters as tab, carriage return, and backspace into an output stream. An escape sequence has the form:

```
►►──\──┬─escape_sequence_character─┬──►◄
       ├─octal_digits──────────────┤
       └─x──hexadecimal_digits─────┘
```

The C language escape sequences and the characters they represent are listed in Table 3 on page 10.

You can place an escape sequence in a character constant or in a string constant. If an escape sequence is not recognized, a warning message is issued. The program compiles, but the compiler removes the backslash (\) and uses the character following it.

Table 3. Escape Sequences

Escape Sequence	Character Represented
`\a`	Alert (bell)
`\b`	Backspace
`\f`	Form feed (new page)
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\v`	Vertical tab
`\'`	Single quotation mark
`\"`	Double quotation mark
`\?`	Question mark
`\\`	Backslash

Note: The line continuation sequence (\ followed by a newline character), which is used in C language character strings to indicate that the current line continues on the next line, is not an escape sequence. See "Preprocessor Directive Format" on page 147 for more information on the line continuation character.

Value

The value of an escape sequence represents the member of the character set used at run time. For example, on a system that uses the ASCII character codes, the letter `V` is represented by the escape sequence `\x56`. On a system using EBCDIC character codes, the letter `V` is represented by `\xE5`.

In string and character sequences, when you want the backslash to represent itself (rather than the beginning of an escape sequence), you must use a \\ backslash escape sequence.

Related Information

- "Character Constants" on page 26
- "String Constants" on page 28

Comments

You can use *comments* to document code. Comments are notes in the source code that are replaced by one space character when the code is compiled. Comments are otherwise ignored.

Comments begin with the `/*` characters, end with the `*/` characters, and can span more than one line. You can place comments anywhere the C language allows white space. White space includes space, tab, form feed, and newline characters.

Note: The `/*` or `*/` characters found in a character constant or string literal do not start or end comments.

In the following program, line 6 is a comment:

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      printf("This program has a comment.\n");
6      /* printf("This is a comment line and will not print.\n");  */
7   }
```

Because the comment on line 6 is equivalent to a space, the output of this program is:

```
This program has a comment.
```

Note that `/*...*/` found in a string literal is not interpreted as a comment. For example, line 6 in the following program is not a comment.

```
1   #include <stdio.h>
2
3   int main(void)
4   {
5      printf("This program does not have \
6   /* NOT A COMMENT */ a comment.\n");
7   }
```

The output of the program is:

```
This program does not have /* NOT A COMMENT */ a comment.
```

You cannot nest comments. Each comment ends at the first occurrence of `*/`.

In the following example, the comments are shaded:

```
1    /* A program with nested comments. */
2
3   #include <stdio.h>
4
5   int main(void)
6   {
7      test_function();
8   }
9
10   int test_function(void)
11   {
12      int number;
13      char letter;
14    /*
15    number = 55;
16    letter = 'A';
17    /* number = 44; */
18   */
19   return 999;
20   }
```

In `test_function()`, the compiler reads the `/*` in line 14 through the `*/` in line 17 as a comment and line 18 as C language code, causing errors at line 18. To avoid commenting over comments already in the source code, you can use conditional compilation preprocessor directives to cause the compiler to bypass sections of a C program.

Note: Under C/VSE, if you use the `SSCOM` option, the compiler recognizes two slashes (//) as the beginning of a comment that terminates at the end of the line. For more information on the `SSCOM` option, refer to the *C/VSE User's Guide*.

Multibyte characters can also be included within a comment.

Keywords

The C language reserves some words for special usage, known as *keywords*. You cannot use them as identifiers. Although you can use them for macro names, it is not recommended that you do so. Only the exact spellings of the words as specified below are reserved. For example, `auto` is reserved but `AUTO` is not.

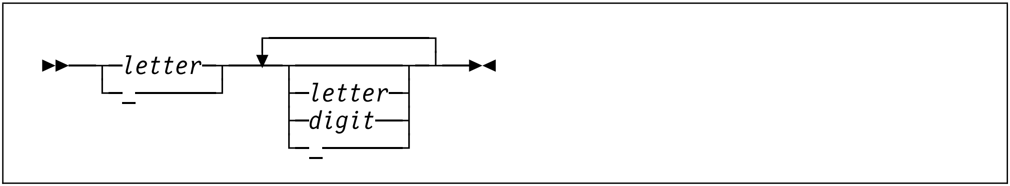
The following table lists the C language keywords:

Table 4. The C Language Keywords

`_Packed`	`double`	`int`	`struct`
`auto`	`else`	`long`	`switch`
`break`	`enum`	`register`	`typedef`
`case`	`extern`	`return`	`union`
`char`	`float`	`short`	`unsigned`
`const`	`for`	`signed`	`void`
`continue`	`goto`	`sizeof`	`volatile`
`default`	`if`	`static`	`while`
`do`			

Identifiers

Identifiers provide names for functions, data objects, labels, tags, parameters, macros, and typedefs. For more information on name spaces, see "Name Spaces" on page 17. An identifier has the form:



There is no limit to the number of characters in an identifier. However, only the first several characters of identifiers are significant. The following table shows the minimal character lengths of identifiers that are recognized. Some compilers may allow longer identifiers.

Identifier	Minimum Number of Significant Characters
Static data objects	255
Static function names	255
External data objects	8
External function names	8

For identifiers, uppercase and lowercase letters are viewed as different symbols. Thus, `PROFIT` and `profit` represent different data objects.

Note: If you do not use the prelinker and long name support provided for the C/VSE compiler, a message will be emitted if you use both `STOCKONHOLD` and `stockonhold` as external identifiers. The linker will interpret `STOCKONHOLD` and `stockonhold` as the same external data object `STOCKONH`. For more information on longname support, see “longname” on page 171. For more information on the prelinker, see the *LE/VSE Programming Guide*. Also see “External Name Mapping in C/VSE” on page 18 and “Long Name Support in C/VSE” on page 19. For complete portability, never use different case representations to refer to the same object.

Do not create identifiers that begin with an underscore (_) for function names and variable names because they may conflict with C/VSE internal names.

Although the names of system calls and library functions are not reserved words, if you do not include the appropriate header files, avoid using them as identifiers. Duplication of a predefined name can lead to confusion for the maintainers of your code and can cause errors at link time or run time. If you include a library in a program, be aware of the function names in that library to avoid name duplications. For more information on identifiers, refer to “External Name Mapping in C/VSE” on page 18, and the *LE/VSE C Run-Time Library Reference*.

You should always include the appropriate header files when using standard library functions.

Scope

An *identifier* is the name used to designate a function or data object. An identifier becomes *visible* with its declaration. Identifiers are described on page 12.

The region where an identifier is visible is referred to as the identifier’s *scope*. The four kinds of scope are:

- Block
- Function
- File
- Function prototype

The scope of an identifier is determined by where the identifier is declared.

Block scope The declaration of the identifier is located inside a block. A block starts with an opening brace ({) and ends with a closing brace (}). An identifier with block scope is visible from the point where it is declared to the closing brace that ends the block.

You can nest block visibility. A block nested inside a block can contain declarations that redeclare variables declared in the outer block. The new declaration of the variable applies to the inner block. The original declaration is restored when program control returns to the outer block. A variable from the outer block is visible inside inner blocks that do not redefine the variable.

Function scope The only identifier with function scope is a label name. A label is implicitly declared by its appearance in the program text. A `goto` statement transfers control to the label specified on the `goto` statement. The label is visible to any `goto` statement that appears in the same function as the label.

File scope
: The declaration of the identifier appears outside of any block. It is visible from the point where it is declared to the end of the source file. If source files are included by `#include` preprocessor directives, those files are considered to be part of the source and the identifier will be visible to all included files that appear after the declaration of the identifier. The identifier can be declared again as a block scope variable. The new declaration replaces the file-scope declaration until the end of the block.

Function prototype scope
: The declaration of the identifier appears within the list of parameters in a function prototype. It is visible from the point where it is declared to the closing parenthesis of the prototype declaration.

In the following example, the variable `x`, defined on line 1, is different from the `x` defined on line 2. The variable defined on line 2 has function prototype scope and is visible only up to the closing parenthesis of the prototype declaration. Visibility of the variable `x` defined on line 1 resumes after the end of the prototype declaration.

```
1   int x = 4;            /* variable x defined with file scope */
2   long myfunc(int x, long y); /* variable x has function      */
3                               /* prototype scope              */
4   int main(void)
5   {
6      /* . . . */
7   }
```

Functions with `static` storage class are visible only in the source file in which they are defined. All other functions can be globally visible. For more information on static storage class, see "static Storage Class Specifier" on page 42.

The program in Figure 1 on page 15 illustrates blocks, nesting, and scope. The example shows two kinds of scope: file and block. Assuming that the function `printf()` is defined elsewhere, the `main()` function prints the values `1, 2, 3, 0, 3, 2, 1` on separate lines. Each instance of `i` represents a different variable.

```
int i = 1;                        /* i defined at file scope */

int main(int argc, char * argv[])
{

   printf("%d\n", i);             /* Prints 1 */

   {
      int i = 2, j = 3;           /* i and j defined at
                                     block scope */
      printf("%d\n%d\n", i, j);   /* Prints 2, 3 */

      {
         int i = 0;               /* i is redefined in a nested block     */
                                  /* previous definitions of i are hidden */
         printf("%d\n%d\n", i, j); /* Prints 0, 3 */
      }

      printf("%d\n", i);          /* Prints 2 */

   }

   printf("%d\n", i);             /* Prints 1 */

}
```

Figure 1. Example of Blocks, Nesting, and Scope

Linkage

The association or lack of association between two identical identifiers is known as *linkage*. A C identifier can have one of the following kinds of linkage:

Internal linkage
: Identical identifiers within a single source file refer to the same data object or function.

External linkage
: Identical identifiers in separately compiled files refer to the same data object or function.

No linkage
: Each identifier refers to a unique object.

In Figure 2, the variable `b` is declared in Source File 2 as `extern` and refers to the same data object as in Source File 1. It has external linkage.

If the declaration of an identifier with file scope contains the keyword `static`, it has *internal linkage*. In Figure 2, all references to the variable `a` in Source File 1 refer to the same data object. The variable `a` in Source File 2 refers to a different data object than `a` in Source File 1.

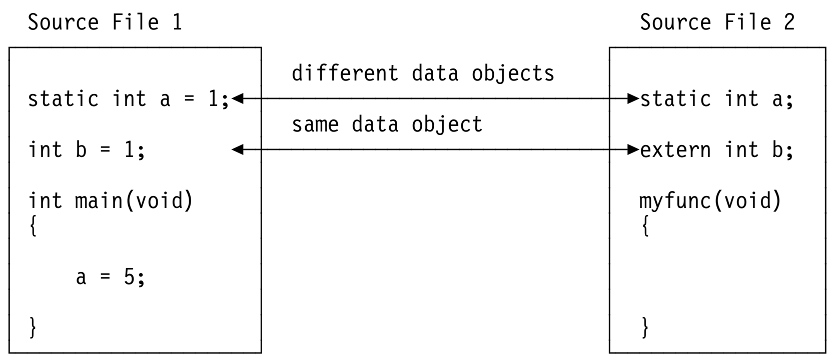


Figure 2. Example of External and Internal Linkage

If the declaration of an identifier has the keyword `extern` and if there is a previous declaration of the identifier at file scope, the identifier has the same linkage as the first declaration. If a definition of the identifier is not visible within the file scope, the identifier has external linkage. In Figure 3, the variable `x` has internal linkage because the first declaration of `x` occurs in file `try.h` and the storage class `static` is specified. The variable `y` in Figure 3 has external linkage because a previous declaration of the identifier `y` is not visible within the same file scope.

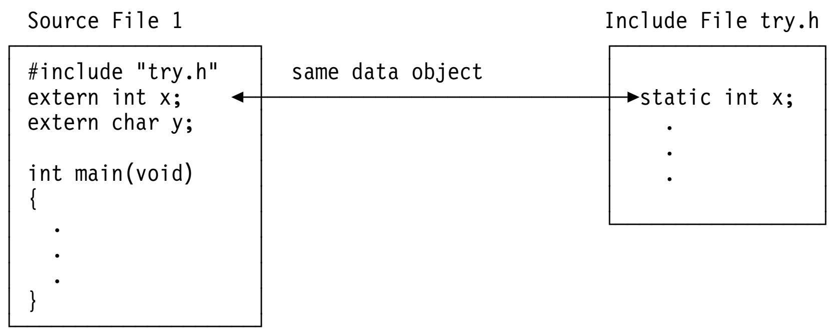


Figure 3. Example of Linkage Using the Keyword extern

If an identifier is declared without a storage class specifier at file scope, it has external linkage.

An identifier that falls into one of the following categories has no linkage:

- An identifier that does not represent an object or a function. For example, a C label is neither an object nor a function.
- An identifier that represents a function parameter.
- An identifier declared inside a block without the keyword `extern`.

You can make identifiers refer to the same object or function in other source files with appropriate `extern` declarations, as described in Chapter 3, “Declarations and Definitions” on page 31.

Storage Duration

Storage duration determines how long storage for an object exists. An object has either *static* storage duration or *automatic* storage class depending on its declaration.

An object with static storage duration has storage allocated for it at initialization; storage remains available until program termination. All objects with file scope have static storage duration. An object has static storage duration if it has internal or external linkage or if it contains the keyword `static`. All other objects have automatic storage.

Storage for an object with automatic storage class is allocated and removed according to the scope of the identifier. For example, storage for an object declared at block scope is allocated when the identifier is declared and removed when the closing brace of the block is reached. An object has automatic storage duration if it is declared with no linkage and does not have the `static` storage class specifier.

Name Spaces

In any C program, identifiers refer to functions, data objects, labels, tags, parameters, macros, and typedefs. The same identifier can be used for more than one class of identifier, as long as you follow the rules outlined in this section.

Name spaces are categories used to group similar types of identifiers.

You must assign unique names within each name space to avoid conflict. The same identifier can be used to declare different objects as long as each identifier is unique within its name space. The context of an identifier within a program lets the compiler resolve its name space without ambiguity.

Identifiers in the same name space can be redefined within enclosed program blocks as described in “Scope” on page 13.

Within each of the following four categories of name spaces, the identifiers must be unique.

- These identifiers must be unique within a single scope:
  - Function names
  - Variable names
  - Names of function parameters
  - Enumeration constants
  - Type definition names
- Tags of these types must be unique within a single scope:
  - Enumerations
  - Structures
  - Unions
- Members of structures and unions must be unique within a single structure or union.
- Statement labels have function scope and must be unique within a function.

Structure tags, structure members, and variable names are in three different name spaces; no conflict occurs among the three items named `student` in the following example:

```
struct student       /*  structure tag       */
{
   char student[20];  /*  structure member    */
   int class;
   int id;
} student;            /*  structure variable  */
```

Each occurrence of `student` is interpreted by its context in the program. For example, when `student` appears after the keyword `struct`, it is a structure tag. When `student` appears after either of the member selection operators **.** or ->, the name refers to the structure member. (See Chapter 4, "Expressions and Operators" on page 93 to find out how to refer to members of union or structure variables.) In other contexts, the identifier `student` refers to the structure variable.

External Name Mapping in C/VSE

The names of variables or functions used in source code that has external linkage are mapped to names used in the object module. When you compile a program with C/VSE, refer to the following as a guide for using names of variables or functions with external linkage:

- Do not use names of the library functions for user-defined functions.
- Some functions in the C run-time environment begin with two underscores (_ _). Do not use an underscore as the first letter of an identifier.
- Each _ is mapped to @ for external names, except when a program is being compiled with the `LONGNAME` compile-time option, in which case the _ remains as an _.
- Three sets of environment functions have names beginning with `IBM`, `CEE`, and `PLI`; avoid using these names. To prevent conflicts between run-time functions and user-defined names, the C/VSE compiler changes all `static` or `extern` variable names that begin with `IBM`, `CEE`, and `PLI` (in your C source program) to `IB$`, `CE$`, and `PL$`, respectively in the object module. If you are using interlanguage calls, avoid using these prefixes because the compiler of the calling or called language may or may not change these prefixes in the same manner as C/VSE. All of this is completely integrated into the C/VSE compiler and Debug Tool for VSE/ESA, so it is not apparent.

  To call an external program or access an external variable that begins with `IBM`, `CEE`, and `PLI`, use the `#pragma map` preprocessor directive. The following is an example of `#pragma map` forcing an external name to be `IBMENTRY`.

  ```
  #pragma map(ibmentry,"IBMENTRY")
  ```

  For more information on the `#pragma map` directive, see "map" on page 171.

Long Name Support in C/VSE

When programs are compiled with C/VSE, the compiler, by default, maps _ to @, truncates external names to 8 characters, and changes them to uppercase because linkage editors support external names only up to a maximum of 8 characters. For example, when the following program is compiled with no options:

```
int test_name[4] = { 4, 8, 9, 10 };
int test_namesum;

int main(void) {
  int i;

  for (i = 0; i < 4; i++)
    test_namesum += test_name[i];
  printf("sum is %d\n", test_namesum);
}
```

The following message is displayed:

```
ERROR EDC0248 External name TEST_NAM cannot be redefined.
```

The external names `test_namesum` and `test_name` are changed to uppercase and truncated to 8 characters. If the `CHECKOUT` compile-time option is specified, the compiler generates two warning messages to this effect:

```
WARNING EDC0244 External name test_namesum has been truncated to TEST_NAM.
WARNING EDC0244 External name test_name has been truncated to TEST_NAM.
```

Because the truncated names are now the same, the compiler produces the `EDC0248` error message and terminates the compilation. The program will not compile.

If the previous program is compiled with the `LONGNAME` compile-time option, no warning or error messages are produced. However, the additional step of prelinking the program is required.

The `LONGNAME` compile-time option supports mixed case external names of up to 255 characters.

Object modules produced by compiling with `LONGNAME` options have external names that are L-names (Long names). Object modules produced by compiling with `NOLONGNAME` options have external names that are S-names (Short names). L-names are mixed case and up to 255 characters long; S-names are uppercase and up to 8 characters long.

There are two alternatives if you want to use external names longer than 8 characters in your source code:

- Use the `#pragma map` to map long external names in the source code to 8 characters or fewer in the object module.

  #pragma map(*verylongname*,"*sname*")

- Use the long name support provided by the compile-time option `LONGNAME`. To use the long name support, you must:

  – Use the `LONGNAME` compile-time option when compiling your program

  – Use the prelinker before creating an executable module. See the *LE/VSE Programming Guide* for more information on the prelinker.

Types

A *type* is the interpretation of a value which can be stored in an object or returned by a function call.

Values stored in objects or returned by functions are accessed by expressions. The meaning of these values is determined by the types associated with each expression.

Types may be divided into three categories: object types (which fully describe objects), function types (which describe functions), and incomplete types (which describe objects, without including their sizes).

Object Types

The supported object types are:

char type
: Defaults to `unsigned char` and is capable of storing any member of a basic character set (see "Character Set" on page 7)

signed integer types
: `signed char`, `signed short int`, `signed int`, `signed long int`

unsigned integer types
: `unsigned char`, `unsigned short int`, `unsigned int`, `unsigned long int`

floating point types
: `float`, `double`, `long double`

fixed-point decimal types
: `decimal` with the length and number of positions after the decimal point specified (this type is an extension to the ANSI standard, and is only available when `LANGLVL` is set to `EXTENDED`)

enumeration type
: Specifies a set of constant values of type `int`

array type
: Specifies a set of objects, each of the same type, which are contiguously allocated

structure type
: Specifies a set of objects of possibly distinct types which are allocated sequentially

union type
: Describes a set of overlapping objects of possibly different types

pointer type
: Describes an object whose value is used to access another object or invoke a function

Function Types

Function types describe a segment of code that is characterized by the number and type of parameters passed to it, and the type of value returned by it.

Incomplete Types

The incomplete types are:

void Can never be completed

array of unknown size
Can be completed by the later declaration of the same identifier with its size specified

structure or union of unspecified content
Can be completed by later declaration of the same structure or union tag with its content defined

For example, the following are incomplete types:

```
struct struct_type *p;
extern int a[];
```

void is an incomplete type that cannot be completed.

An array of unknown size is completed for an identifier of that type by specifying the size in a later declaration.

A structure or union of unknown content is completed for all declarations of that type by declaring the same structure or union tag with its content later in the same scope.

The following is an example of completing a structure type that was incomplete when it was first declared:

```
struct employee;

struct employee
  {
  char *name;
  int age;
  int salary;
  } company[200];
```

Constants

The C language contains the following types of constants:

- Integer
- Floating-point
- Fixed-point decimal
- Character
- String
- Enumeration

A constant is data with a value that does not change during the execution of a program. The value of any constant must be in the range of representable values for its type.

Note: Any plus or minus unary operator sign preceding a constant expression is not part of the constant expression.

For more information on data types, see “Type Specifiers” on page 44.

Integer Constants

Integer constants can have one of the following values:

- Decimal
- Octal
- Hexadecimal

They have these forms:

```
►►──┬─decimal_constant─────┬──┬─────────────────┬──►◄
    ├─octal_constant───────┤  ├─┬─l─┬──┬─────┬─┤
    └─hexadecimal_constant─┘  │ └─L─┘  ├─u─┤ │
                              │        └─U─┘ │
                              └─┬─u─┬──┬─────┬─┘
                                └─U─┘  ├─l─┤
                                       └─L─┘
```

Data Type

The data type of an integer constant is determined by the constant’s value. Table 5 describes the integer constant and a list of possible data types for that constant. The first data type in the list that can contain the constant value will be associated with the constant.

Table 5. Data Types for Integer Constants

Constant	Data Type
unsuffixed decimal	`int, long int, unsigned long int`
unsuffixed octal	`int, unsigned int, long int, unsigned long int`
unsuffixed hexadecimal	`int, unsigned int, long int, unsigned long int`
suffixed by `u` or `U`	`unsigned int, unsigned long int`
suffixed by `l` or `L`	`long int, unsigned long int`
suffixed by both `u` or `U`, and `l` or `L`	`unsigned long int`

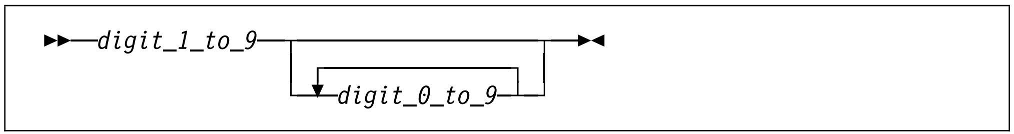
A plus (+) or minus (-) symbol can precede the constant. It is treated as a unary operator rather than as part of the constant value.

Related Information

- “Decimal Constants” on page 23
- “Octal Constants” on page 24
- “Hexadecimal Constants” on page 23
- “Integers” on page 47

Decimal Constants

A *decimal constant* contains any of the digits `0` through `9`. A decimal constant has the form:



Note: The first digit cannot be `0`. An integer constant beginning with the digit `0` is interpreted as an octal constant, rather than as a decimal constant.

Data Type

See Table 5 on page 22 for a complete description of the data types of decimal constants.

Examples

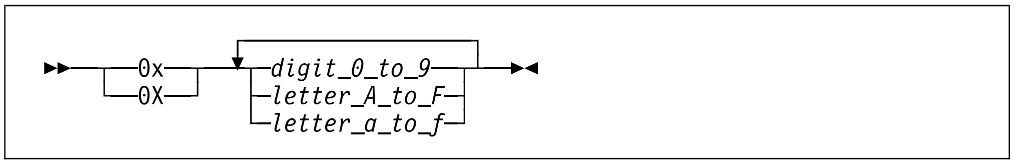
```
485976
433132211
20
5
```

Related Information

- “Integer Constants” on page 22
- “Octal Constants” on page 24
- “Hexadecimal Constants” on page 23
- “Integers” on page 47

Hexadecimal Constants

A *hexadecimal constant* begins with the `0` digit followed by either an `x` or `X`. After the `0x` or `0X`, you can place any combination of the digits `0` through `9` and the letters `a` through `f` or `A` through `F`. When used to represent a hexadecimal constant, the lowercase letters are equivalent to their corresponding uppercase letters. A hexadecimal constant has the form:



Data Type

See Table 5 on page 22 for a complete description of the data types of hexadecimal constants.

Examples

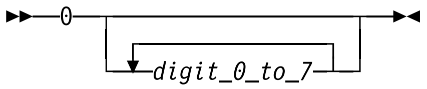
```
0x3b24
0XF96
0x21
0x3AA
0X29b
0X4bD
```

Related Information

- “Integer Constants” on page 22
- “Decimal Constants” on page 23
- “Octal Constants” on page 24
- “Integers” on page 47

Octal Constants

An *octal constant* begins with the digit `0` and contains any of the digits `0` through `7`. An octal constant has the form:



Data Type

See Table 5 on page 22 for a complete description of the data types of octal constants.

Examples

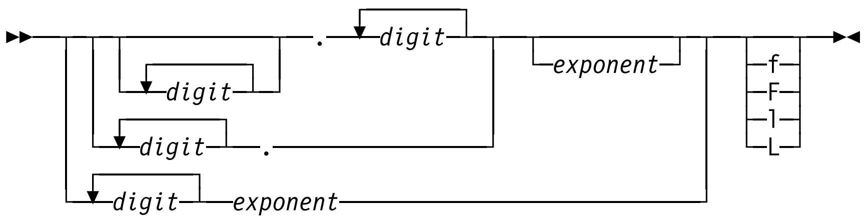
```
0
0125
034673
03245
```

Related Information

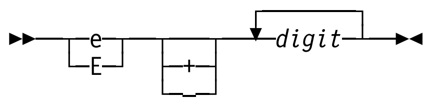
- “Integer Constants” on page 22
- “Decimal Constants” on page 23
- “Hexadecimal Constants” on page 23
- “Integers” on page 47

Floating-Point Constants

A *floating-point constant* consists of an integral part, a decimal point, a fractional part, an exponent part, and an optional suffix. Both the integral and fractional parts are made up of decimal digits. You can omit either the integral part or the fractional part, but not both. You can omit either the decimal point or the exponent part, but not both.
A floating-point constant has the form:



The exponent part consists of `e` or `E`, followed optionally by a sign and a decimal number. An exponent has the form:



Value

The floating-point constant `8.45e+3` evaluates as follows:

8.45 * 10^3 = 8450.0

The representation of a floating-point number on a system is unspecified by the standards. If a floating-point constant is too large or too small, the result is undefined.

Data Type

The suffix `f` or `F` indicates a type of `float`, and the suffix `l` or `L` indicates a type of `long double`. If a suffix is not specified, the floating-point constant has a type `double`.

A plus (+) or minus (-) symbol can precede a floating-point constant. However, it is not part of the constant; it is interpreted as a unary operator.

Examples

Floating-Point Constant	**Value**
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100,000
7.321E-3	0.007321
3.2E+4	32,000
0.5e-6	0.0000005
0.45	0.45
6.e10	60,000,000,000

Related Information

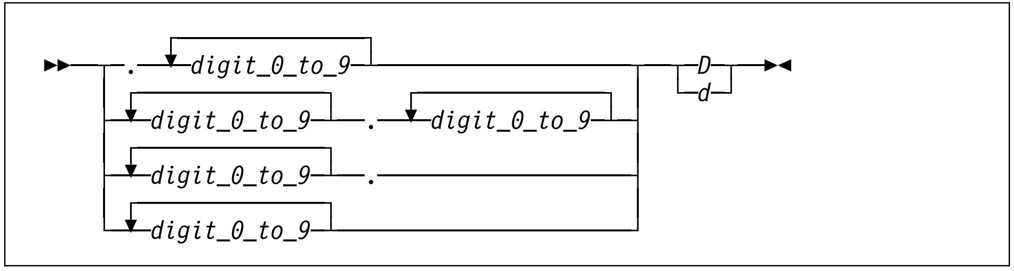
- "Floating-Point" on page 45

Fixed-Point Decimal Constants

Fixed-point decimal constants are a C/VSE extension to ANSI C. This type is available when `LANGLVL` is set to `EXTENDED`.

A *fixed-point decimal constant* has a numeric part and a suffix that specifies its type. The numeric part can include a digit sequence representing the whole-number part, followed by a decimal point (.), followed by a digit sequence representing the fraction part. Either the integral part or the fractional part, or both must be present.

A fixed-point constant has the form:



A fixed-point constant has two attributes:

Number of digits (size)
Number of decimal places (precision)

Data Type

The suffix `D` or `d` indicates a fixed-point constant.

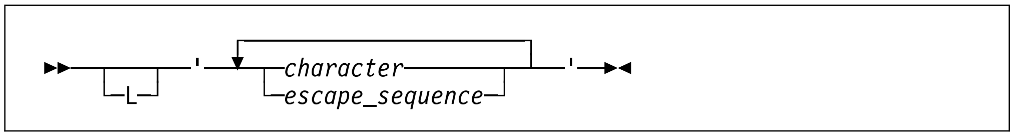
Examples

Fixed-Point Constant	(size, precision)
`1234567890123456D`	`(16,  0)`
`12345678.12345678D`	`(16,  8)`
`12345678.d`	`( 8,  0)`
`.1234567890d`	`(10, 10)`
`12345.99d`	`( 7,  2)`
`000123.990d`	`( 9,  3)`
`0.00D`	`( 3,  2)`

For more information on fixed-point decimal data types, see the *LE/VSE C Run-Time Programming Guide*.

Character Constants

A *character constant* consists of a sequence of characters or escape sequences enclosed in single quotation marks. A character constant has the form:



At least one character or escape sequence must appear in the character constant. The prefix `L` indicates a wide-character constant; no prefix indicates an integer character constant. A character constant must appear on a single source line.

The character constant can contain any character from the C character set, with the following restrictions:

Single quotation mark
Use the backslash followed by a single quotation mark `\'` to represent the single quotation mark.

Newline character
Use the backslash followed by the newline character `\n` to represent the newline escape sequence.

Backslash character
Use the backslash followed by a second backslash \\ to represent the backslash escape sequence.

Value

The value of a character constant containing a single character is the numeric representation of the character in the character set used at run time. The value of a wide-character constant containing a single multibyte character is the code for that character, as defined by the `mbtowc` function. If the character constant contains more than one character, the last 4 bytes represent the character constant.

Data Type

A character constant has type `int`. A wide-character constant is represented by a double-byte character of type `wchar_t`, as defined in the `<stddef.h>` include file. Multibyte characters represent character sets that go beyond the single-byte character set. Each multibyte character can contain up to 4 bytes.

Examples

```
'a'    '\''
'0'    '('
'x'    '\n'
'7'    '\117'
'C'
```

The following example, using character constants and wide-character constants, shows what happens when the character constant contains more than one character:

EDCXRAAD

```
 /* Example of how to use character constants */

#include <stdio.h>
#include <stdlib.h>

char ch1 = 'd';
char ch2 = 'abcd';
char ch3 = '\204';
char ch4 = '\x84';
wchar_t wch1 = L'd';
wchar_t wch2 = L'abcd';
wchar_t wch3 = L'\204';
wchar_t wch4 = L'\x84';
```

```
int main(void)
  {
  printf("ch1  = %c, ch2  = %c\n", ch1, ch2);
  printf("ch3  = %c, ch4  = %c\n", ch3, ch4);
  printf("wch1 = %lc, wch2 = %lc\n", wch1, wch2);
  printf("wch3 = %lc, wch4 = %lc\n", wch3, wch4);
  }
```

This example produces the following output:

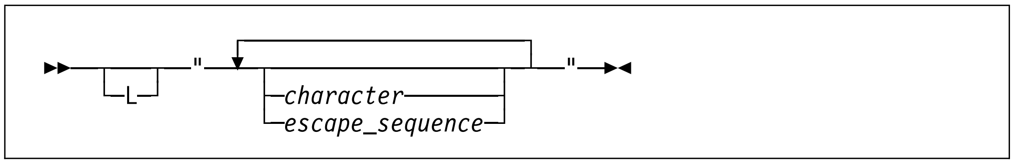
```
ch1  = d, ch2  = d
ch3  = d, ch4  = d
wch1 = d, wch2 = d
wch3 = d, wch4 = d
```

Related Information

- “String Constants” on page 28
- “Escape Sequences” on page 9
- “Integers” on page 47

String Constants

A *string constant* or *literal* contains a sequence of characters or escape sequences enclosed in double quotation marks. A string constant has the form:



The prefix L indicates a wide-character string literal; no prefix indicates a character string literal.

A null (\0) character is appended to each string. For a wide-character string, the value 0 of type wchar_t is appended. By convention, programs recognize the end of a string by finding the null character.

The string constant can contain any character from the C character set, with the following restrictions:

Double quotation mark
Use the escape sequence \" to represent the double quotation mark. You can represent the single quotation mark by itself '.

Newline character
Use the escape sequence \n to represent a newline character as part of the string.

Another way to continue a string is to have two or more consecutive strings. Adjacent string literals are concatenated to produce a single string. The null character of the first string will no longer exist after the concatenation. You cannot concatenate a wide-string constant with a character-string constant.

Multiple spaces contained within a string constant are retained.

Backslash character
Use the escape sequence \\ to represent a backslash character as part of the string.

Note: When you modify string literals, the resulting behavior depends on whether your strings are stored in writable static. See “#pragma” on page 162 for more information on `#pragma strings` that can be used to specify whether your string literals are read-only or writable. String literals are writable by default.

Data Type

A character string constant has type *array of* `char` and static storage duration. A wide character constant has type *array of* `wchar_t` and static storage duration.

```
char titles[ ] = "Bach’s \"Jesu, Joy of Man's Desiring\"";
char *mail_addr = "Last Name    First Name    MI   Street Address   \
   City     Province   Postal code ";
char *temp_string = "abc" "def" "ghi";  /* *temp_string = "abcdefghi\0" */
```

Examples

The following example shows string literals used in the `strcpy()`, `strcat()`, and `printf()` functions.

EDCXRAAE

```
 /* Example of how to use string literals in functions */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char ch1[20];
char ch2[20] = "Example!!";

int main(void)
  {
  strcpy(ch1, "My ");
  strcat(ch1, ch2);
  printf("ch1 = %s\n", ch1);
  }
```

This example produces the following output:

```
ch1 = My Example!!
```

Related Information

- “Character Constants” on page 26
- “Escape Sequences” on page 9
- “Characters” on page 44
- “Arrays” on page 71

Enumeration Constants

When you define an enumeration data type, you specify a set of identifiers that the data type represents. Each identifier in this set is an *enumeration constant*.

Value

Each enumeration constant has an integer value. You can use an enumeration constant anywhere an integer constant is allowed. The value of the constant is determined in the following way:

1. An equal sign (=) and a constant expression after the enumeration constant give an explicit value to the constant. The identifier represents the value of the constant expression.

2. If no explicit value is assigned, the leftmost constant in the list receives the value zero (0).
3. Identifiers with no explicitly assigned values receive the integer value that is one greater than the value represented by the previous identifier.

Data Type

An enumeration constant has type `int`.

Examples

The following data type declarations list `oats`, `wheat`, `barley`, `corn`, and `rice` as enumeration constants. The number under each constant shows the integer value.

```
enum grain { oats, wheat, barley, corn, rice };
   /*         0      1      2      3     4          */

enum grain { oats=1, wheat, barley, corn, rice };
   /*         1        2      3      4     5        */

enum grain { oats, wheat=10, barley, corn=20, rice };
   /*          0     10        11     20       21    */
```

It is possible to associate the same integer with two different enumeration constants. For example, the following definition is valid. The identifiers `suspend` and `hold` have the same integer value.

```
enum status { run, delete=5, suspend, resume, hold=6 };
   /*          0      5        6       7      6   */
```

Related Information

- “Enumerations” on page 63
- “Integers” on page 47

Chapter 3. Declarations and Definitions

This chapter describes the C language definitions and declarations for data objects and functions. A *declaration* establishes the names and characteristics of data objects and functions used in a program. A *definition* is a declaration that allocates storage for data objects or specifies the body for a function.

The following table shows examples of declarations and definitions. The identifiers declared in the first column do not allocate storage; they refer to a corresponding definition. The identifiers declared in the second column allocate storage; they are both declarations and definitions.

Table 6. Examples of Declarations and Definitions

Declarations	Declarations and Definitions
`extern double pi;`	`double pi = 3.14159265;`
`float square(float x);`	`float square(float x) { return x*x; }`
`struct payroll;`	`struct payroll { char *name; float salary; } employee;`

Function declarations are described in “Function Declarations” on page 89.

Declarations

A declaration introduces an identifier and specifies the following properties associated with it:

- Scope, which describes the visibility of an identifier in a block or source file. See “Scope” on page 13.
- Storage duration, which describes when storage for a data object is allocated and freed.
- Linkage, which describes the association between two identical identifiers. See “Linkage” on page 15 for more information.
- Type, which describes the kind of data the object is to represent.

In C, the lexical order of elements of declaration is as follows:

- Storage duration and linkage specification, described in “Storage Class Specifiers” on page 33
- Type qualifiers and qualifiers, described in “Qualifiers” on page 67
- Type specification, described in “Type Specifiers” on page 44
- Declarators, which introduce identifiers and derived types such as arrays, pointers, and functions, described in “Declarators” on page 70
- Initializers, which initialize storage with initial values, described in “Initializers” on page 91.

The positioning of declarations in the source code determines the scope or visibility and possibly the assumed storage duration and linkage of the identifier.

Block Scope Data Declarations

A *block scope data declaration* can only be placed at the beginning of a block. It declares a variable and makes that variable accessible to the current block. All block scope declarations that do not have the `extern` storage class specifier are definitions and allocate storage for that object.

You can declare a data object with block scope to any one of the following storage class specifiers:

- `auto`
- `register`
- `static`
- `extern`

If you do not specify a storage class specifier in a block-scope data declaration, the default storage class specifier `auto` is used. If you specify a storage class specifier, you can omit the type specifier. If you omit the type specifier, all variables declared in that declaration will have the type `int`.

Initialization of Variables

You cannot initialize a variable that is declared in a block scope data declaration and has the `extern` storage class specifier.

The types of variables you can initialize and the values that uninitialized variables receive vary for each storage class specifier.

Storage for Objects

Declarations with the `auto` or `register` storage class specifier result in automatic storage duration. Declarations with the `extern` or `static` storage class specifier result in static storage duration.

Related Information

- “auto Storage Class Specifier” on page 34
- “static Storage Class Specifier” on page 42
- “Declarators” on page 70
- “Initializers” on page 91
- “Type Specifiers” on page 44

File Scope Data Declarations

A *file scope data declaration* appears outside any block. It describes a variable and makes that variable accessible to all functions that are in the same file and whose definitions appear after the declaration.

A *file scope data definition* is a data declaration at file scope that also causes the system to allocate storage for that variable. All objects whose identifiers are declared at file scope have static storage duration.

You can use a file scope data declaration to declare variables that you want several functions to access.

The only storage class specifiers you can place in a file scope data declaration are `static` and `extern`. If you specify `static`, all variables declared in it have internal linkage. If you do not specify `static`, all variables declared in it have external linkage.

If you specify the storage class `static` or `extern`, you can omit the type specifier. If you omit the type specifier, all variables defined in that declaration receive the type `int`.

Initialization of Variables

You can initialize any object with file scope. If you do not initialize a file scope variable, its initial value is zero of the appropriate type. If you do initialize it, the initializer must be described by a constant expression, or it must reduce to the address of a previously declared variable at file scope, possibly modified by a constant expression. Initialization of all variables at file scope takes place before the `main()` function begins execution.

Storage for Objects

All objects with file scope data declarations have static storage duration. The system allocates memory for all file scope variables when the program begins execution and frees it when the program is finished executing.

Related Information

- "extern Storage Class Specifier" on page 36
- "static Storage Class Specifier" on page 42
- "Declarators" on page 70
- "Initializers" on page 91
- "Type Specifiers" on page 44

Storage Class Specifiers

This section describes C language object declarations, the storage durations associated with the objects, and the linkage of their identifiers. The storage class specifier used within the declaration determines the following:

- Whether the object has internal, external, or no linkage.
- Whether the storage duration of the object is static class (storage for the object is maintained throughout program execution) or automatic class (storage for the object is maintained only during the execution of the block in which the identifier of the object is defined).
- Whether the object is to be stored in memory or in a register, if available.
- Whether the object is initialized with the default value of zero of the appropriate type or an indeterminate default initial value.

For a function, the storage class specifier determines the function's linkage.

auto Storage Class Specifier

The `auto` storage class specifier enables you to define a variable with automatic storage; its use and storage are restricted to the current block. The storage class keyword `auto` is optional in a data declaration and forbidden in a parameter declaration. A variable having the `auto` storage class specifier must be declared within a block. It cannot be used for file scope declarations.

The following example lines declare variables having the `auto` storage class specifier:

```
auto int counter;
auto char letter = 'k';
```

Initialization of Variables

You can initialize any `auto` variable except parameters. If you do not initialize an automatic object, its value is undefined. If you provide an initial value, the expression representing the initial value can be any valid C expression. For aggregates or unions, the initial value must be a valid constant expression. The object is then set to that initial value each time the program block that contains the object's definition is entered.

Note: If you use the `goto` statement to jump into the middle of a block, automatic variables within that block are not initialized.

Storage for Objects

Objects with the `auto` storage class specifier have automatic storage duration. Each time a block is entered, storage for `auto` objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If an `auto` object is defined within a function that is recursively invoked, memory is allocated for the object at each invocation of the block.

Usage

Declaring variables with the `auto` storage class specifier can decrease the amount of memory required for program execution, because `auto` variables require storage only while they actually are needed.

Examples

The following program shows the scope and initialization of `auto` variables. The function `main()` defines two variables, each named `auto_var`. The first definition occurs on line 8. The second definition occurs in a nested block on line 11. While the nested block is executed, only the `auto_var` created by the second definition is available. During the rest of the program, only the `auto_var` created by the first definition is available.

EDCXRAAF

```
 /* Example of how to use auto variables */

#include <stdio.h>

int main(void)
{
   void call_func(int passed_var);
   auto int auto_var = 1; /* first definition of auto_var  */

   {
      int auto_var = 2;   /* second definition of auto_var */
      printf("inner auto_var = %d\n", auto_var);
   }
   call_func(auto_var);
   printf("outer auto_var = %d\n", auto_var);
}

void call_func(int passed_var)
{
   printf("passed_var = %d\n", passed_var);
   passed_var = 3;
   printf("passed_var = %d\n", passed_var);
}
```

This example produces the following output:

```
inner auto_var = 2
passed_var = 1
passed_var = 3
outer auto_var = 1
```

The following example uses an array that has the storage class `auto` to pass a character string to the function `sort()`. The C language views an array name that appears without subscripts (for example, `string`, instead of `string[0]`) as a pointer. Thus, `sort()` receives the address of the character string, rather than the contents of the array. The address enables `sort()` to change the values of the elements in the array.

EDCXRAAG

```
 /* Example of how to pass an array name to a function
    This example sorts a string */

#include <stdio.h>

int main(void)
{
   void sort(char *array, int n);
   char string[75];
   int length;
   printf("Enter letters:\n");
   scanf("%74s", string);
   length = strlen(string);
   sort(string,length);
   printf("The sorted string is: %s\n", string);
}

void sort(char *array, int n)
{
   int gap, i, j, temp;

   for (gap = n / 2; gap > 0; gap /= 2)
      for (i = gap; i < n; i++)
         for (j = i - gap; j >= 0 && array[j] > array[j + gap];
            j -= gap)
         {
            temp = array[j];
            array[j] = array[j + gap];
            array[j + gap] = temp;
         }
}
```

When the program is run, interaction with the program could produce:

Output `Enter letters:`

Input `zyfab`

Output `The sorted string is:  abfyz`

Related Information

- "register Storage Class Specifier" on page 41
- "Block Scope Data Declarations" on page 32
- "Address &" on page 104

extern Storage Class Specifier

The `extern` storage class specifier enables you to declare objects and functions that several source files can use. All object declarations that occur outside a function and that do not contain a storage class specifier declare identifiers with external linkage. All function definitions that do not specify a storage class define functions with external linkage.

You can distinguish an `extern` declaration from an `extern` definition by the presence of the keyword `extern` and the absence of an initializer. If the keyword `extern` is absent or if there is an initializer, the declaration is also a definition; otherwise, it is just a declaration. An `extern` definition can appear only outside a function definition. Only one definition of an external variable is allowed, and that declaration is the definition of the storage for the variable.

If a declaration for an identifier already exists at file scope, any `extern` declaration of the same identifier found within a block refers to that same object. If no other declaration for the identifier exists at file scope, the identifier has external linkage.

Declaration

An `extern` declaration can appear outside a function or at the beginning of a block. If the declaration describes a function or appears outside a function and describes an object with external linkage, the keyword `extern` is optional.

If you choose not to specify a storage class specifier, the function will have external linkage. So if you include a declaration for the same function with the storage class specifier `static` before the declaration with no storage class specifier, an error will be noted because of the incompatible declarations. If you had included the `extern` storage class specifier on the original declaration, there would be no error and the function would have external linkage.

Initialization of Variables

You can initialize any object with the `extern` storage class specifier at file scope. You can initialize an `extern` object with an initializer that must either:

- Appear as part of the definition. The initial value must be described by a constant expression.
- Reduce to the address of a previously declared object with static storage duration. This object may be modified by a constant expression.

If you do not initialize an `extern` variable, its initial value is zero of the appropriate type. Initialization of an `extern` object is completed by the start of program execution.

Storage for Objects

`extern` objects have static storage duration. Memory is allocated for `extern` objects before the `main()` function begins execution. When the program finishes executing, the storage is freed.

Controlling External Static

Certain program variables with the `extern` storage class may be constant and never written to. If this is the case, it is not necessary to have a copy of these variables made for every user of the program. In addition, there may be a need to share constant program variables between C and another language.

extern Example 1

The following program fragment shows how to force an external program variable to be part of the program that includes executable code and constant data by using the `#pragma variable(varname, NORENT)` directive:

```
#pragma options(RENT)

#pragma variable(rates, NORENT)
extern float rates[5] = { 3.2, 83.3, 13.4, 3.6, 5.0 };

extern float totals[5];

int main(void) {
⋮
}
```

In this example, the source file is to be compiled with the `RENT` option. The variable `rates` is included with the executable code because `#pragma variable(rates, NORENT)` has been specified. The variable `totals` is included with the writable static. Each user has his/her own copy of the array totals, and the array rates is shared between all users of the program. This sharing may yield a performance and storage benefit.

The `#pragma variable(varname, NORENT)` does not apply to, and has no effect on, program variables with the `static` storage class. Program variables with the `static` storage class are always included with the writable static. An informational message appears if you write to a nonreentrant variable when the `CHECKOUT` compile-time option is specified.

When `#pragma variable(varname, NORENT)` is specified for a variable, care must be taken to ensure that this variable is never written to. Program exceptions or unpredictable program behavior may result should this be the case. In addition, `#pragma variable(varname, NORENT)` must be included in every source file where the variable is referenced or defined.

For more information on the `RENT` and `NORENT` compile-time options, refer to the *C/VSE User's Guide*.

Example 2

The following program shows the linkage of `extern` objects and functions. The `extern` object `total` is declared on line 11 of `File 1` and on line 10 of `File 2`. The definition of the external object `total` appears in `File 3`. The `extern` function `tally()` is defined in `File 2`. The function `tally()` can be placed in the same file as `main()` or in a different file. Because `main()` precedes these definitions and uses both `total()` and `tally()`, `main()` declares `tally()` on line 11 and `total()` on line 12.

EDCXRAH1

```
 /* Example shows the linkage of extern objects/functions
    File 1 of 3 - other files are EDCXRAH2, EDCXRAH3
    In this file, the program receives the price of an item,
    adds the tax, and prints the total cost of the item */

#include <stdio.h>

int main(void)
{   /* begin main */
   void tally(void);   /* declaration of function tally */
   extern float total; /* first declaration of total    */

   printf("Enter the purchase amount: \n");
   tally();
   printf("\nWith tax, the total is:  %.2f\n", total);
}   /* end main */
```

EDCXRAH2

```
 /* Example shows the linkage of extern objects/functions
    File 2 of 3 - other files are EDCXRAH1, EDCXRAH3
    this file defines the function tally */

#define  tax_rate  0.05

void tally(void)
{   /* begin tally */
   float tax;
   extern float total; /* second declaration of total       */

   scanf("%f", &total);
   tax = tax_rate * total;
   total += tax;
}   /* end tally */
```

EDCXRAH3

```
 /* Example shows the linkage of extern objects/functions
    File 3 of 3 - other files are EDCXRAH1, EDCXRAH2 */
float total;
```

The following program shows `extern` variables used by two functions. Because both functions `main()` and `sort()` can access and change the values of the `extern` variables `string` and `length`, `main()` does not have to pass parameters to `sort()`.

EDCXRAAI

```
 /* Example shows how extern variables are used by two functions */

#include <stdio.h>

char string[75];
int length;

int main(void)
{
   void sort(void);

   printf("Enter letters:\n");
   scanf("%s", string);
   length = strlen(string);
   sort();
   printf("The sorted string is: %s\n", string);
}

void sort(void)
{
   int gap, i, j, temp;

   for (gap = length / 2; gap > 0; gap /= 2)
      for (i = gap; i < length; i++)
         for (j = i - gap;
              j >= 0 && string[j] > string[j + gap];
              j -= gap)
         {
            temp = string[j];
            string[j] = string[j + gap];
            string[j + gap] = temp;
         }
}
```

When this program is run, interaction with the previous program could produce:

Output	`Enter letters:`
Input	`zyfab`
Output	`The sorted string is:  abfyz`

The following program shows a `static` variable `var1` that is defined at file scope and then declared with the storage class specifier `extern`. The second declaration refers to the first definition of `var1` and so it has internal linkage.

```
static int var1;
⋮
extern int var1;
```

Related Information

- “File Scope Data Declarations” on page 32
- “Function Definition” on page 83
- “Function Declarator” on page 85
- “Constant Expression” on page 96

register Storage Class Specifier

The `register` storage class specifier indicates to the compiler within a block scope data definition or a parameter declaration that the object being described will be heavily used (such as a loop control variable). The compiler may use this information to place the object into a machine register for fast access storage. The storage class keyword `register` is required in a data definition and in a parameter declaration that describes an object having the `register` storage class. An object having the `register` storage class specifier must be defined within a block or declared as a parameter to a function.

The following example lines define automatic storage duration objects using the `register` storage class specifier:

```
register int score1 = 0, score2 = 0;
register unsigned char code = 'A';
register int *element = &order[0];
```

Initialization of Variables

You can initialize any `register` auto storage objects except parameters.

Storage for Objects

Objects with the `register` storage class specifier have automatic storage duration. Each time a block is entered, storage for `register` objects defined in that block is made available. When the block is exited, the objects are no longer available for use.

If a `register` object is defined within a function that is recursively invoked, the system allocates memory for the variable at each invocation of the block.

The `register` storage class specifier indicates that the object is heavily used and indicates to the compiler that the value of the object should reside in a machine register. Because of the limited size and number of registers available on most systems, few variables can be stored in registers at the same time.

If the compiler does not allocate a machine register for a `register` object, the object is treated as having the storage class specifier `auto`. In C programs, even if a `register` variable is treated as a variable with storage class `auto`, the address of the variable cannot be taken.

Restrictions

You cannot apply the `&` (address) operator to `register` variables.

You cannot use the `register` storage class specifier on file scope data declarations.

Usage

The compiler treats the register attribute as a suggestion under the `OPTIMIZE(1)` optimization level. Under `OPTIMIZE(0)`, the register attribute is ignored. For more information on improving code quality using optimization, refer to the *LE/VSE C Run-Time Programming Guide*.

Related Information

- “auto Storage Class Specifier” on page 34
- “Block Scope Data Declarations” on page 32
- “Function Declarator” on page 85
- “Address &” on page 104

static Storage Class Specifier

The `static` storage class specifier enables you to define objects with static storage duration and internal linkage, or to define functions with internal linkage.

An object having the `static` storage class specifier can be defined within a block or at file scope. If the definition occurs within a block, the object has no linkage. If the definition occurs at file scope, the object has internal linkage.

Initialization of Variables

You can initialize any `static` object. If you do not provide an initial value, the object receives the value of zero of the appropriate type. If you initialize a `static` object, the initializer must be described by a constant expression or must reduce to the address of a previously declared `extern` or `static` object, possibly modified by a constant expression.

Storage for Objects

Objects with the `static` storage class specifier have static storage duration. The storage for a `static` variable is made available when the program begins execution. When the program finishes executing, the memory is freed.

Restrictions

You cannot declare a `static` function at block scope.

Usage

You can use `static` variables when you need an object that retains its value from one execution of a block to the next execution of that block. Using the `static` storage class specifier keeps the system from reinitializing the object each time the block in which the object is defined is executed.

Examples

The following program shows the linkage of `static` identifiers at file scope. This program uses two different external `static` identifiers named `stat_var`. The first definition occurs in `File 1`. The second definition occurs in `File 2`. The `main()` function references the object defined in `File 1`. The `var_print()` function references the object defined in `File 2`.

EDCXRAJ1

```
 /* Example of how to use file scope static variables
    File 1 of 2 - other file is EDCXRAJ2 */

#include <stdio.h>
extern void var_print(void);
static stat_var = 1;

int main(void)
{
   printf("file1 stat_var = %d\n", stat_var);
   var_print();
   printf("FILE1 stat_var = %d\n", stat_var);
}
```

EDCXRAJ2

```
 /* Example of how to use file scope static variables
    File 2 of 2 - other file is EDCXRAJ1 */

static int stat_var = 2;

void var_print(void)
{
    printf("file2 stat_var = %d\n", stat_var);
}
```

This example produces the following output:

```
file1 stat_var = 1
file2 stat_var = 2
FILE1 stat_var = 1
```

The following program shows the linkage of `static` identifiers with block scope. The function `test()` defines the `static` variable `stat_var`. `stat_var` retains its storage throughout the program, even though `test()` is the only function that can reference `stat_var`.

EDCXRAAK

```
 /* Example of how to use block scope static variables */

#include <stdio.h>

int main(void)
{
   void test(void);
   int counter;
   for (counter = 1; counter <= 4; ++counter)
      test();
}
```

```
void test(void)
{
   static int stat_var = 0;
   auto int auto_var = 0;
   stat_var++;
   auto_var++;
   printf("stat_var = %d auto_var = %d\n", stat_var, auto_var);
}
```

This example produces the following output:

```
stat_var = 1 auto_var = 1
stat_var = 2 auto_var = 1
stat_var = 3 auto_var = 1
stat_var = 4 auto_var = 1
```

Related Information

- “Block Scope Data Declarations” on page 32
- “File Scope Data Declarations” on page 32
- “Function Definition” on page 83
- “Function Declarator” on page 85

Type Specifiers

The C data types are:

- Characters
- Floating-point
- Fixed-point decimals
- Integers
- Void
- Structures
- Unions
- Enumerations

From these types, you can derive the following:

- Arrays
- Pointers
- Functions

Characters

The C language has three character data types: `char`, `signed char`, and `unsigned char`. These data types provide enough storage to hold any member of the character set used at run time.

`char` is represented as an `unsigned char`. For information on changing this default, see “chars” on page 165. If it does not matter whether a `char` data object is `signed` or `unsigned`, you can declare the object as having the data type `char`; otherwise, explicitly declare `signed char` or `unsigned char`. When a `char` (`signed` or `unsigned`) is widened to an `int`, its value is preserved.

To declare a data object having a character data type, place a *char specifier* in the type specifier position of the declaration. The `char` specifier has the form:

```
►►──┬───────────┬──char──►◄
    ├─unsigned──┤
    └─signed────┘
```

The declarator for a simple character declaration is an identifier. You can initialize a simple character with a character constant or with an expression that evaluates to an integer.

Examples

The following example defines the identifier `end_of_string` as a constant object of type `char` having the initial value `\0` (the null character):

```
const char end_of_string = '\0';
```

The following example defines the `unsigned char` variable `switches` as having the initial value `3`:

```
unsigned char switches = 3;
```

You can use the `char` specifier in variable definitions to define such variables as arrays of characters, pointers to characters, and arrays of pointers to characters.

The following example defines `string_pointer` as a pointer to a character:

```
char *string_pointer;
```

The following example defines `name` as a pointer to a character. After initialization, `name` points to the first letter in the character string `"Johnny"`:

```
char *name = "Johnny";
```

The following example defines a one-dimensional array of pointers to characters. The array has three elements. Initially they are a pointer to the string `"Venus"`, a pointer to `"Jupiter"`, and a pointer to `"Saturn"`:

```
static char *planets[ ] = { "Venus", "Jupiter", "Saturn" };
```

Related Information

- “Arrays” on page 71
- “Pointers” on page 78
- “Character Constants” on page 26
- “Assignment Expression” on page 114

Floating-Point

The C language defines three types of floating-point variables: `float`, `double`, and `long double`. They are defined in the header file `float.h`.

In C/VSE, a float occupies 4 bytes in storage, a double occupies 8 bytes, and a long double occupies 16 bytes. Thus, the following expression always evaluates to `1` (true):

```
sizeof(float) <= sizeof(double) && sizeof(double) <= sizeof(long double)
```

To declare a data object having a floating-point type, use the *float specifier*. The float specifier has the form:

```
►►──┬─float───────┬──►◄
    ├─double──────┤
    └─long double─┘
```

The declarator for a simple floating-point declaration is an identifier. You can initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. (The storage class of a variable determines how you can initialize the variable.)

Examples

The following example defines the identifier `pi` for an object of type `double`:

```
double pi;
```

The following example defines the `float` variable `real_number` with the initial value `100.55`: The float constant `100.55` is of type double, but using it to initialize `float` variable `real_number` converts it to a float.

```
static float real_number = 100.55;
```

The following example defines the `float` variable `float_var` with the initial value `0.0143`:

```
float float_var = 1.43e-2;
```

The following example declares the `long double` variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` with 20 elements of type `double`:

```
double table[20];
```

Related Information

- “Floating-Point Constants” on page 24
- “Assignment Expression” on page 114
- “Integers” on page 47
- “Floating Point” on page 179

Fixed-Point Decimal Data Types

Use the type specifier *decimal(n,p)* to declare fixed-point decimal variables and to initialize them with fixed-point decimal constants. *decimal* is a macro defined in `decimal.h`. Remember to include `decimal.h` if you use fixed-point decimals in your program.

Fixed-point decimal types are classified as arithmetic types. *decimal(n,p)* designates a decimal number with *n* digits, and *p* decimal places. *n* is the total number of digits for the integral and decimal parts combined, and *p* is the number of digits for the decimal part only. For example, decimal(5,2) represents a number, such as, 123.45 where *n*=5 and *p*=2. The value for *p* is optional. If it is left out, the default value is `0`.

n and *p* have a range of allowed values according to the following rules:

```
p ≤ n
1 ≤ n ≤ DEC_DIG
0 ≤ p ≤ DEC_PRECISION
```

Note: DEC_DIG (the maximum number of digits *n*) and DEC_PRECISION (the maximum precision *p*) are defined in `decimal.h`. Currently, a maximum of 31 digits is used for both limits.

The following example shows how you can declare a variable as a fixed-point decimal data type:

```
decimal(10,2)  x;
decimal(5,0)   y;
decimal(5)     z;
decimal(18,10) *ptr;
decimal(8,2)   arr[100];
```

In the previous example:

- *x* can have values between -99999999.99D and +99999999.99D.
- *y* and *z* can have values between -99999D and +99999D.
- *ptr* is a pointer to type decimal(18,10).
- *arr* is an array of 100 elements, where each element is of type decimal(8,2).

The syntax for the fixed-point decimal type specifier is as follows:

```
►►──decimal──(──constant_expression──┬─────────────────────────┬──)──►◄
                                     └─,constant_expression────┘
```

The constant expression is evaluated as a positive integral constant expression. The second constant expression is optional. If it is left out, the default value is 0. decimal(*n*,*0*) and decimal(*n*) are type compatible.

Integers

C/VSE supports six types of integer variables:

- `short int`, `short`, `signed short`, or `signed short int`
- `signed int` or `int` (In some cases, no type specifier is needed; see “Block Scope Data Declarations” on page 32 and “File Scope Data Declarations” on page 32.)
- `long int`, `long`, `signed long`, or `signed long int`
- `unsigned short int` or `unsigned short`
- `unsigned` or `unsigned int`
- `unsigned long int` or `unsigned long`

The storage size of a `short` type is less than or equal to the storage size of an `int` variable, and the storage size of an `int` variable is less than or equal to the storage size of a `long` variable. Thus, the following expression always evaluates to `1` (true):

```
sizeof(short) <= sizeof(int) && sizeof(int) <= sizeof(long)
```

Two sizes for integer data types are provided. Objects having type `short` are 2 bytes of storage long. Objects having type `long` are 4 bytes of storage long. An `int` represents the most efficient data storage size on the system (the word-size of the machine) and receives 4 bytes of storage.

The `unsigned` prefix indicates that the value of the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, `int` reserves the same storage as `unsigned int`. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

To declare a data object having an integer data type, place an *int specifier* in the type specifier position of the declaration. The `int` specifier has the form:

```
►►──┬──────────┬──┬──┬───────┬──int──┬──►◄
    ├─unsigned─┤  │  ├─short─┤       │
    └─signed───┘  │  └─long──┘       │
                  ├─short────────────┤
                  └─long─────────────┘
```

The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. (The storage class of a variable determines how you can initialize the variable.)

Examples

The following example defines the `short int` variable `flag`:

```
short int flag;
```

The following example defines the `int` variable `result`:

```
int result;
```

The following example defines the `unsigned long int` variable `ss_number` as having the initial value `438888834`:

```
unsigned long ss_number = 438888834ul;
```

The following example defines the identifier `sum` for an object of type `int`. The initial value of `sum` is the result of the expression `a + b`:

```
extern int a, b;
auto sum  = a + b;
```

Related Information

- "Integer Constants" on page 22
- "Decimal Constants" on page 23
- "Octal Constants" on page 24
- "Hexadecimal Constants" on page 23

void Type

`void` is a data type that always represents an empty set of values. The keyword for this type is `void`. When a function does not return a value, use `void` as the type specifier in the function definition and declaration. The only object that can be declared with the type specifier `void` is a pointer.

Example

In line 5 of the following example, the function `find_max()` is declared as having type `void`. Lines 14 through 23 contain the complete definition of `find_max()`.

Note: The use of the `sizeof` operator in line 11 is a standard method of determining the number of elements in an array.

EDCXRAAM

```
 1    /* Example use of void data type */
 2
 3   #include <stdio.h>
 4    /* declaration of function find_max */
 5   extern void find_max(int x[ ], int j);
 6
 7   int main(void)
 8   {
 9      static int numbers[ ] = { 99, 54, -102, 89 };
10
11      find_max(numbers, (sizeof(numbers) / sizeof(numbers[0])));
12   }
13   void find_max(int x[ ], int j)
14   { /* begin definition of function find_max */
15      int i, temp = x[0];
16
17      for (i = 1; i < j; i++)
18      {
19          if (x[i] > temp)
20             temp = x[i];
21      }
22      printf("max number = %d\n", temp);
23   } /* end definition of function find_max  */
```

Related Information

- “Cast” on page 104
- “Integers” on page 178

Structures

A *structure* contains an ordered group of data objects. Unlike the elements of an array, the data objects within a structure can have varied data types. Each data object in a structure is a *member* or *field*.

You can use structures to group logically related objects. For example, if you want to allocate storage for the components of one address, you can define the following variables:

```
int street_no;
char *street_name;
char *city;
char *prov;
char *postal_code;
```

If you want to allocate storage for more than one address, however, you can group the components of each address by defining a structure data type and defining several variables having the structure data type:

```
 1   struct address {
 2                    int street_no;
 3                    char *street_name;
 4                    char *city;
 5                    char *prov;
 6                    char *postal_code;
 7                  };
 8   struct address perm_address;
 9   struct address temp_address;
10   struct address *p_perm_address = &perm_address;
```

Lines 1 through 7 declare the structure tag `address`. Line 8 defines the variable `perm_address`, and line 9 defines the variable `temp_address`, both of which are instances of the structure `address`. Both `perm_address` and `temp_address` contain the members described in lines 2 through 6. Line 10 defines a pointer `p_perm_address`, which points to a structure of type `address`. `p_perm_address` is initialized to point to `perm_address`.

You can reference a member of a structure by specifying the structure variable name with the **.** (period) or a pointer to a struct with the -> and the member name. For example, both of the following:

```
perm_address.prov = "Ontario";
p_perm_address -> prov = "Ontario";
```

assign a pointer to the string `"Ontario"` to the pointer `prov` that is in the structure `perm_address`.

All references to structures must be fully qualified. Therefore, in the preceding example, you cannot reference the fourth field by `prov` alone. You must reference this field by `perm_address.prov`.

You cannot declare a structure with members of incomplete types.

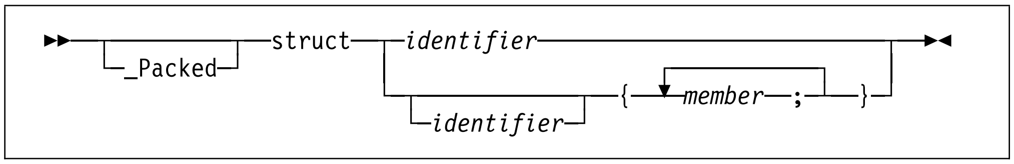
Declaring a Structure Data Type

A structure type declaration does not allocate storage. It describes the members that are part of the structure.

You can declare structures having any storage class. The C/VSE compiler treats structures declared with the `register` storage class specifier as automatic structures.

A structure type declaration contains the `struct` keyword followed by an optional identifier (the structure tag) and a brace-enclosed list of members.

A structure declaration has the form:

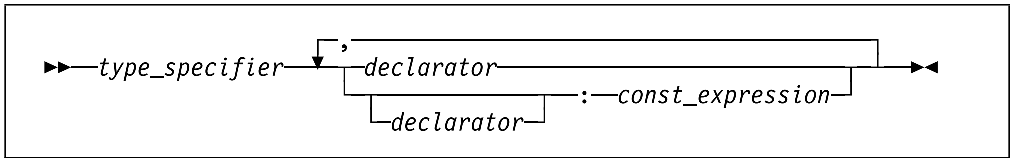


The keyword struct followed by the identifier (tag) names the data type. If you do not provide a tag, you must place all variable definitions that refer to that data type within the statement that defines the data type.

A structure variable definition contains a storage class keyword, the struct keyword, a structure tag, a declarator, and an optional identifier. The structure tag indicates the data type of the structure variable.

The list of members provides the data type with a description of the values that can be stored in the structure.

A member has the form:



A member that does not represent a bit field can be of any data type and can have the volatile or const qualifier. For more information on qualifiers, refer to “Qualifiers” on page 67. If a **:** (colon) and a constant expression follow the declarator, the member represents a *bit field*. For more information on bit fields, refer to “Declaring and Using Bit Fields” on page 58.

Identifiers used as aggregate or member names can be redefined to represent different objects in the same scope without conflicting. You cannot use the name of a member more than once in a structure type, but you can use the same member name in another structure type that is defined within the same scope.

You cannot declare a structure type that contains itself as a member but you can declare a structure type that contains a pointer to itself as a member.

Example of Declaring a Structure Type and Structure Variables

You can place a type definition and a variable declaration in one statement by placing a declarator and an initializer (optional) after the type definition. If you want to specify a storage class specifier for the variable, you must place the storage class specifier at the beginning of the statement. For example:

```
static struct {
                int street_no;
                char *street_name;
                char *city;
                char *prov;
                char *postal_code;
             } perm_address, temp_address;
```

The preceding example does not name the structure data type. Thus, perm_address and temp_address are the only structure variables that will have this data type. If an identifier is placed after struct, additional variable definitions of this data type can be made later in the program.

The structure type (or tag) cannot have the volatile qualifier, but a member or a structure variable can be defined as having the volatile qualifier.

For example:

```
static struct class1 {
                        char descript[20];
                        volatile long code;
                        short complete;
                     } volatile file1, file2;
struct class1 subfile;
```

This example gives the `volatile` qualifier to the structures `file1` and `file2`, and to the structure member `subfile.code`.

Initializers of Structures

The initializer contains an `=` (equal sign) followed by a brace-enclosed comma-separated list of values. You do not have to initialize all members of a structure.

The following definition shows a completely initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
static struct address perm_address =
               { 9876, "Goto St.", "Cville", "Ontario", "X9X 1A1"};
```

The values of `perm_address` are:

Member	**Value**
`perm_address.street_no`	`9876`
`perm_address.street_name`	address of string `"Goto St."`
`perm_address.city`	address of string `"Cville"`
`perm_address.prov`	address of string `"Ontario"`
`perm_address.postal_code`	address of string `"X9X 1A1"`

The following definition shows a partially initialized structure:

```
struct address {
                 int street_no;
                 char *street_name;
                 char *city;
                 char *prov;
                 char *postal_code;
               };
struct address temp_address =
               { 321, "Aggregate Ave.", "Structown", "Ontario" };
```

The values of `temp_address` are:

Member	**Value**
`temp_address.street_no`	`321`
`temp_address.street_name`	address of string `"Aggregate Ave."`
`temp_address.city`	address of string `"Structown"`
`temp_address.prov`	address of string `"Ontario"`
`temp_address.postal_code`	depends on the storage class (see note below)

Note: The initial value of `temp_address.postal_code` depends on the storage class associated with the member. See "Storage Class Specifiers" on page 33 for details on the initialization of different storage classes.

The following is an example of using an abstract data type `ControlBlock` without knowing the details of the internals of the data type.

The advantage of this style of programming is that as long as the interface to the data type remains the same, the code that uses that interface does not have to change if the internal structure of the data type changes.

`cblock.c` defines the data type `ControlBlock` and the functions that are the interface to `ControlBlock`. The internal structure of `ControlBlock` and the interface functions are not declared.

```
 /* Example of an abstract data type
    File 1 of 2 - other file is cblock.c */

#include "cblock.c"

main()
{
  ControlBlock* ctl;

  ctl = GetControlBlock();     /* allocate and return a control block */

  UpdateControlBlock(ctl, 27); /* update the control block with 27    */

  UseControlBlock(ctl);        /* use the updated control block      */
}
```

```
 /* cblock.c - Example of an abstract data type
    File 2 of 2 */

#ifndef __INTERFACE__
  #define __INTERFACE__

  typedef struct ControlBlock_T ControlBlock;

  ControlBlock* GetControlBlock();
  int UpdateControlBlock(ControlBlock*, int);
  int UseControlBlock(ControlBlock*);

#endif
```

Example

The following program finds the sum of the integer numbers in a linked list.

EDCXRAAS

```
 /* Example of a linked list */

#include <stdio.h>

struct record {
                int number;
                struct record *next_num;
              };

int main(void)
{
   struct  record name1, name2, name3;
   struct  record *recd_pointer = &name1;
   int sum = 0;

   name1.number = 144;
   name2.number = 203;
   name3.number = 488;

   name1.next_num = &name2;
   name2.next_num = &name3;
   name3.next_num = NULL;

   while (recd_pointer != NULL)
   {
      sum += recd_pointer->number;
      recd_pointer = recd_pointer->next_num;
   }
   printf("Sum = %d\n", sum);
}
```

The structure type `record` contains two members: `number` (an integer) and `next_num` (a pointer to a structure variable of type `record`).

The `record` type variables `name1`, `name2`, and `name3` are assigned the following values:

Member Name	**Value**
`name1.number`	`144`
`name1.next_num`	The address of `name2`
`name2.number`	`203`
`name2.next_num`	The address of `name3`
`name3.number`	`488`
`name3.next_num`	`NULL` (Indicating the end of the linked list)

The variable `recd_pointer` is a pointer to a structure of type `record`. `recd_pointer` is initialized to the address of `name1` (the beginning of the linked list).

The `while` loop causes the linked list to be scanned until `recd_pointer` equals `NULL`. This statement advances the pointer to the next object in the list:

```
recd_pointer = recd_pointer->next_num;
```

Packed Structures

C/VSE supports both packed (`_Packed` attribute) and unpacked structures.

Data elements of a structure are stored in memory on an address boundary specific for that data type. For example, a `double` value is stored in memory on a doubleword (8-byte) boundary. Gaps may be left in memory between elements of a structure to align elements on their natural boundaries. You can reduce the padding of bytes within a structure by using the `_Packed` qualifier on the structure declaration.

Structure Alignment

Unpacked structures are mapped as follows:

1. A structure is placed on the strictest boundary required by any of its elements.
2. Each subsequent data element is placed on its natural boundary.
3. Bit fields are packed starting from bit 0 (the high order bit).

   In the following example, the two integers `x` and `y` take 7 bits in a single byte. The C/VSE compiler places 1 unused bit after the bit field, before the `char` is placed on a byte boundary.

   ```
   struct {
            double   a       ;
            int      x : 3   ;   /* bit field */
            int      y : 4   ;   /* bit field */
            /* there will be 1 unused bit right here */
            char b           ;
          } axyb;
   ```

   In the following example, C/VSE places 25 unused bits after the bit field, before the integer is placed on a fullword boundary.

   ```
   struct {
            double   a       ;
            int      x : 3   ;   /* bit field */
            int      y : 4   ;   /* bit field */
            /* there will be 25 unused bits right here */
            int b            ;
          };
   ```

4. A fullword boundary is forced by a construct:

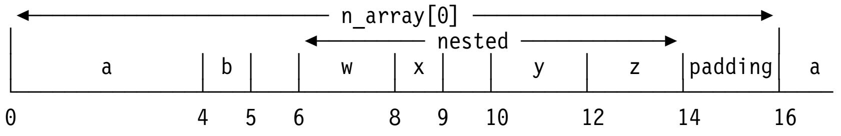
   ```
            int        :0   ;
   ```

Structures and members having the `_Packed` attribute are not always aligned on natural boundaries. Alignment of fields within a structure can be determined by using the `AGGREGATE` compile-time option.

Example 1: The following example illustrates the alignment of structures and structure members in memory for `_Packed` and nonpacked structures.

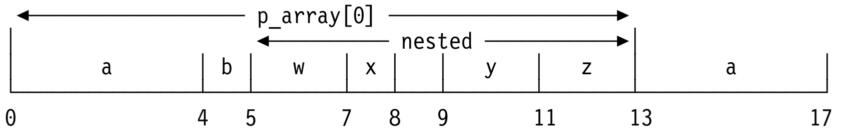
```
struct ss {
             int    a;
             char   b;
             struct {
                       short w;
                       char  x;
                       short y;
                       short z;
                    } nested;
          };

struct ss          n_array[2];
_Packed struct ss  p_array[2];
```

The unqualified array `n_array` maps into memory as follows:



Note: The 6th, 10th, 15th, and 16th bytes are padding.

The packed array, `p_array`, is padded with fewer bytes. The `p_array` has the following organization:



Note: The 9th byte is padding.

In the previous example, the `_Packed` qualifier only applies to the first level of members, and not to elements of imbedded structures. All the members of the external structure `ss` (`a`, `b`, and `nested`) are aligned on byte boundaries. Members of the internal structure `nested` (`w`, `x`, `y`, and `z`) are mapped on their normal nonpacked boundaries. To pack the internal structure `nested`, you must explicitly specify the `_Packed` keyword when `nested` is defined.

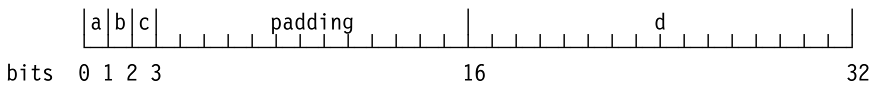
Finally, because this is a packed array, the array elements only have to be aligned on byte boundaries. There is no extra padding at the end of each array element.

Example 2: If the `_Packed` structure contains bit fields, the member following the bit fields always starts on a byte boundary. That is, structure members that are not bit fields never begin in the middle of a byte.

```
struct ss {
             int    a : 1;
             int    b : 1;
             int    c : 1;
             short  d;
          };

struct ss          normal;
_Packed struct ss  packed;
```

The memory layout of the nonpacked structure `normal` is as follows:

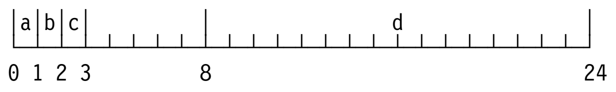
Note: The offsets shown in this map are expressed in bits, not bytes.



Note: The 4th through 16th bits are padding.

The bit-field elements are put into the first three bits of the structure. Because the structure is not packed, the next non-bitfield member, `d`, is aligned on a 2-byte (16-bit) boundary; hence, 13 bits of padding are to the right of `c`.

The memory layout (in bits) of the packed structure `packed` is as follows:



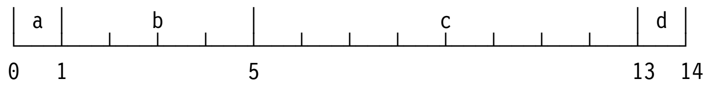
Note: The 4th through 8th bits are padding.

As in the nonpacked structure, the first three members are put into the first three bits of the packed structure. However, because the `_Packed` qualifier was specified, the next non-bitfield member, `d`, is aligned on the next byte boundary, not the next halfword (16-bit) boundary.

Example 3: This example shows the layout of packed structures containing pointers.

```
_Packed struct {
                  char   a;
                  char   *b;
                  double c;
                  char   d;
               } ptrstruct;
```

For C/VSE, pointer alignment is not required. Assuming that pointers are all 32 bits long, the memory layout for `ptrstruct` is:



Because all of the members only have to be aligned on byte boundaries, the alignment requirement of the entire structure becomes 1 byte.

Related Information

- “Structure and Union Member Specification . –>” on page 101
- “Declarators” on page 70
- “Initializers” on page 91
- “C Data Mapping” on page 92

Declaring and Using Bit Fields

A structure can contain *bit fields* that allow you to access individual bits. You can use bit fields for data that requires just a few bits of storage. A bit field declaration contains a type specifier followed by an optional declarator, a colon, a constant expression, and a semicolon. The constant expression specifies how many bits the field reserves. A bit field that is declared as having a length of `0` causes the next field to be aligned on the next integer boundary. For a `_Packed` structure, a bit field of length `0` causes the next field to be aligned on the next byte boundary. Bit-fields with a length of `0` must be unnamed.

For portability, do not use bit fields greater than 32 bits in size.

You cannot define an array of bit fields, or take the address of a bit field.

You can declare a bit field as type `int`, `signed int`, or `unsigned int`. Bit fields of the type `int` default to type `unsigned int`.

If a series of bit fields does not add up to the size of an `int`, padding can take place. The amount of padding is determined by the alignment characteristics of the members of the structure. In some instances, bit fields can cross word boundaries.

The following example declares the identifier `kitchen` to be of type `struct on_off`:

```
struct on_off {
                  unsigned light : 1;
                  unsigned toaster : 1;
                  int count;
                  unsigned ac : 4;
                  unsigned : 4;
                  unsigned clock : 1;
                  unsigned : 0;
                  unsigned flag : 1;
              } kitchen ;
```

The structure `kitchen` contains eight members. The following table describes the storage that each member occupies:

Member Name	Storage Occupied
`light`	1 bit
`toaster`	1 bit
	padding to next `int` boundary
`count`	The size of an `int`
`ac`	4 bits
	4 bits
`clock`	1 bit
	padding to next `int` boundary
`flag`	1 bit
	`flag` will be aligned on the boundary of the next word, because it follows a zero bitfield.
	any unnamed padding as necessary to achieve the appropriate alignment were the structure to be an element of an array

You cannot reference the second field by `toaster`. You must reference this field by `kitchen.toaster`.

The following expression sets the `light` field to `1`:

```
kitchen.light = 1;
```

When you assign to a bit field a value that is out of its range, the bit pattern is preserved and the appropriate bits are assigned. The following expression sets the `toaster` field of the `kitchen` structure to `0` because only the least significant bit is assigned to the `toaster` field:

```
kitchen.toaster = 2;
```

Unions

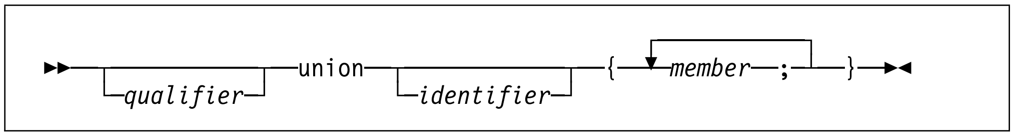
A *union* is an object that can hold any one of a set of named members. The members of the named set can be of any data type. Members are overlaid in storage.

Declaring a Union

The storage allocated for a union is the storage required for the largest member of the union (plus any padding that is required so that the union will end at a natural boundary of its strictest member).

A union type declaration contains the `union` keyword followed by an identifier, which is called the union tag and is optional, and a brace-enclosed list of members.

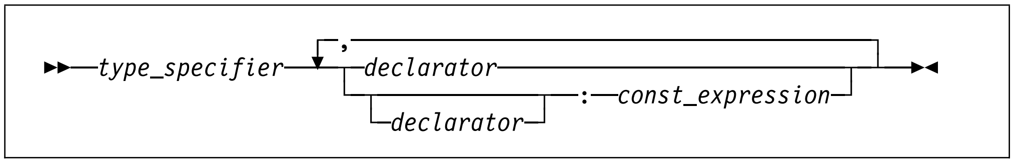
The following diagram shows the form of a union type declaration:



The identifier is a tag given to the union specified by the member list. If you specify a tag, any subsequent declaration of the union (in the same scope) can be made by declaring the tag and omitting the member list. If you do not specify a tag, you must place all variable definitions that refer to that union within the statement that defines the data type.

The list of members provides the data type with a description of the objects that can be stored in the union.

A member has the form:



You can reference one of the possible members of a union as you reference a member of a structure. For example:

```
union {
      char birthday[9];
      int sex:1;  /*  0 = male; 1 = female  */
      float weight;
      } people;

people.birthday[0] = '\n';
```

assigns `'\n'` to the first element in the character array `birthday`, a member of the union `people`. At any given time, a union can represent only one of its members. In the preceding example, the union `people` will contain either `sex`, `birthday`, or `weight` but never more than one of these. For example, the following is not recommended.

```
1  people.birthday = "25/10/67";
2  people.sex = 1;
3  printf("%d\n", people.weight);
```

The assignment on line 2 overwrites the assignment on line 1, so that the printf function will print a meaningless value for `people.weight`.

Example of Defining a Variable That Has Union Data Type

A union variable definition contains the `union` keyword, a union tag, and a declarator. The union tag defines the data type of the union variable.

Type Specifier: The type specifier contains the keyword `union` followed by the name of the union type. You must declare the union data type before you can define a union having that type.

You can define a union data type and a union of that type in the same statement by placing the variable declarator after the data type definition.

Declarator: The declarator is an identifier, possibly with the `volatile` or `const` qualifier.

Initializer: You can initialize only the first member of a union.

The following example shows how you would initialize the first union member `birthday` of the union variable `people`:

```
union {
      char birthday[9];
      int age;
      float weight;
      } people = "04/06/57";
```

Defining a Union Type and a Union Variable

You can place a type definition and a variable definition in one statement by placing a declarator after the type definition. If you want to specify a storage class specifier for the variable, you must place the storage class specifier at the beginning of the statement.

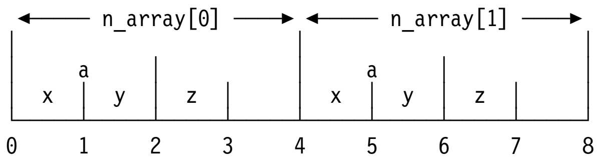
Defining a Packed Union

You can use `_Packed` to qualify a union. However, the memory layout of the union members is not affected. Each member starts at offset zero. The `_Packed` qualifier does affect the total alignment restriction of the whole union. Consider the following example:

```
union uu {
  short    a;
  struct {
    char x;
    char y;
    char z;
  } b;
};

union uu           n_array[2];
_Packed union uu   p_array[2];
```

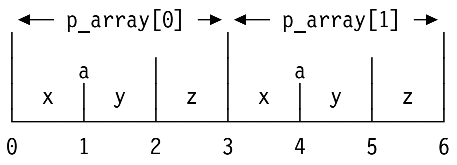
Each of the elements in the nonpacked `n_array` is of type `union uu`. Because the array is nonpacked, each element has an alignment restriction of 2 bytes (the largest alignment requirement among the union members is that of `short a`), and there is 1 byte of padding at the end of each element to enforce this requirement.

The layout in memory is as follows:



Note: The 4th and 8th bytes are padding.

Now consider the packed array `p_array`. Because each of its elements is of type `_Packed union uu`, the alignment restriction of every element is the byte boundary. Therefore, each element has a length of only 3 bytes, instead of the 4 bytes in the previous example.



Examples

The following example defines a union data type (not named) and a union variable (named `length`). The member of `length` can be a `long int`, a `float`, or a `double`.

```
union {
        float meters;
        double centimeters;
        long inches;
      } length;
```

The following example defines the union type `data` as containing one member. The member can be named `charctr`, `whole`, or `real`. The second statement defines two `data` type variables: `input` and `output`.

```
union data {
             char charctr;
             int whole;
             float real;
           };
union data input, output;
```

The following statement assigns a character to `input`:

```
input.charctr = 'h';
```

The following statement assigns a floating-point number to member `output`:

```
output.real = 9.2;
```

The following example defines an array of structures named `records`. Each element of `records` contains three members: the integer `id_num`, the integer `type_of_input`, and the `union` variable `input`. `input` has the `union` data type defined in the previous example.

```
struct {
         int id_num;
         int type_of_input;
         union data input;
       } records[10];
```

The following statement assigns a character to the structure member `input` of the first element of `records`:

```
records[0].input.charctr = 'g';
```

The following shows a struct with a nested enumerated variable and a union:

```
enum possible_shape { circle, square };

struct shape_info {
                  enum possible_shape shape;
                  union {
                          int radius;
                          int diameter;
                        } size;
                  };
```

Related Information

- “Structure and Union Member Specification . –>” on page 101
- “Declarators” on page 70
- “Initializers” on page 91
- “C Data Mapping” on page 92

Enumerations

An *enumeration* data type represents a set of values that you declare. You can define an enumeration data type and all variables that have that enumeration type in one statement, or you can separate the declaration of the enumeration data type from all variable definitions. The identifier associated with the data type (not an object) is a tag. C maps enumeration data items to one of the following minimum applicable types:

- Signed char
- Signed short
- Signed int

Declaring an Enumeration Data Type

An enumeration type declaration contains the `enum` keyword followed by an identifier (the enumeration tag) and a brace-enclosed list of enumerators. Each enumerator is separated by a comma. An enumeration type declaration has the form:

```
                              ┌─,──────────┐
►►──enum──┬────────────┬──{───▼─enumerator─┴───}──►◄
          └─identifier─┘
```

The keyword `enum`, followed by the identifier, names the data type (like the tag on a `struct` data type). The list of enumerators provides the data type with a set of values. Each enumerator represents an integer value. To conserve space, the compiler may store enumerations in spaces smaller than the size of an `int`. An enumerator has the form:

```
►►──identifier──┬──────────────────────────┬──►◄
                └─ = ──constant_expression─┘
```

The identifier in an enumerator is called an *enumeration constant*. You can use it anywhere an integer constant is allowed.
The value of an enumeration constant is determined by the following rules:

1. If an `=` (equal sign) and a constant expression follow the identifier, the identifier represents the value of the constant expression.
2. If the enumerator is the leftmost value in the list, the identifier represents the value `0`.
3. Otherwise, the identifier represents the integer value that is one greater than the value represented by the preceding enumerator.

The following example declares the enumeration tag `status`:

```
enum status { run, create, delete=5, suspend };
   /*        0     1       5       6    */
```

The number under each constant shows the integer value.

Each enumerator identifier must be unique within the block or the file where the enumeration data type is declared. In the following example, the declarations of `average` on line 4 and of `poor` on line 5 cause compiler error messages:

```
 1   func()
 2   {
 3      enum score { poor, average, good };
 4      enum rating { below, average, above };
 5      int poor;
 6   }
```

Example of Defining a Variable That Has an Enumeration Type

An enumeration variable definition contains a storage class specifier (optional), a type specifier, a declarator, and an initializer (optional). The type specifier contains the keyword `enum` followed by the name of the enumeration data type. You must declare the enumeration data type before you can define a variable having that type.

The first line of the following example declares the enumeration tag `grain`. The second line defines the variable `g_food` and gives `g_food` the initial value of `barley` (`2`). The type specifier `enum grain` indicates that the value of `g_food` is a member of the enumerated data type `grain`:

```
enum grain { oats, wheat, barley, corn, rice };
enum grain g_food = barley;
```

The initializer for an enumeration variable contains the `=` symbol followed by an expression. The expression must evaluate to an `int` value.

Example of Defining an Enumeration Type and Enumeration Objects

You can place a type definition and a variable definition in one statement by placing a declarator and an optional initializer after the type definition.
If you want to specify a storage class specifier for the variable, you must place the storage class specifier at the beginning of the declaration. For example:

```
register enum score { poor=1, average, good } rating = good;
```

This example is equivalent to the following two declarations:

```
enum score { poor=1, average, good };
register enum score rating = good;
```

Both examples define the enumeration data type `score` and the variable `rating`. `rating` has the storage class specifier `register`, the data type `enum score`, and the initial value `3` (or `good`).

If you combine a data type definition with the definitions of all variables having that data type, you can leave the data type unnamed. For example:

```
enum { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,
      Saturday } weekday;
```

This example defines the variable `weekday`, which can be assigned any of the specified enumeration constants.

Example

The following program receives an integer as input. The output is a sentence that gives the French name for the weekday that is associated with the integer. If the integer is not associated with a weekday, the program prints `"C'est le mauvais jour."`

EDCXRAAN

```
 /* Example of an enumeration */

#include <stdio.h>

enum days {
            Monday=1, Tuesday, Wednesday,
            Thursday, Friday, Saturday, Sunday
          } weekday;

int main(void)
{
   int num;

   printf("Enter an integer for the day of the week.  "
          "Mon=1,...,Sun=7\n");
   scanf("%d", &num);
   weekday=num;
   french(weekday);
}
```

```
french(weekday)
enum days weekday;
{
   switch (weekday)
   {
      case Monday:
         printf("Le jour de la semaine est lundi.\n");
         break;
      case Tuesday:
         printf("Le jour de la semaine est mardi.\n");
         break;
      case Wednesday:
         printf("Le jour de la semaine est mercredi.\n");
         break;
      case Thursday:
         printf("Le jour de la semaine est jeudi.\n");
         break;
      case Friday:
         printf("Le jour de la semaine est vendredi.\n");
         break;
      case Saturday:
         printf("Le jour de la semaine est samedi.\n");
         break;
      case Sunday:
         printf("Le jour de la semaine est dimanche.\n");
         break;
      default:
         printf("C'est le mauvais jour.\n");
   }
}
```

Related Information

- “Enumeration Constants” on page 29
- “Constant Expression” on page 96
- “Identifiers” on page 12

Tags

Tags uniquely identify *struct*, *union* or *enum* types. Tags can be declared together with the content of the structure or union or with the enumerators list, or they can be declared separately to specify an incomplete type (*struct*, *union*, or *enum*).

Related Information

- “Structures” on page 49
- “Unions” on page 59
- “Enumerations” on page 63

Qualifiers

volatile and const Type Qualifiers

The `volatile` qualifier maintains the intent of the original expression with respect to the order of stores and fetches of `volatile` objects. The `volatile` qualifier is useful for data objects having values that may be changed in ways unknown to your program (such as the system clock). Portions of an expression that reference `volatile` objects are not to be optimized.

The `const` qualifier explicitly declares a data object as a data item that cannot be changed. Its value is set at initialization. You cannot use `const` data objects in expressions requiring a modifiable `lvalue`. For example, a `const` data object cannot appear on the left-hand side of an assignment statement.

For information on lvalues, refer to "Lvalue" on page 95.

For a `volatile` or `const` pointer, you must place the keyword between the `*` and the identifier. For example:

```
int * volatile x;        /* x is a volatile pointer to an int */
int * const y = &z;      /* y is a const pointer to the int variable z */
```

For a pointer to a `volatile` or `const` data object, the type specifier, qualifier, and storage class can be in any order. For example:

```
volatile int *x;         /* x is a pointer to a volatile int  */
or
int volatile *x;         /* x is a pointer to a volatile int  */

const int *y;            /* y is a pointer to a const int     */
or
int const *y;            /* y is a pointer to a const int     */
```

In the following example, the pointer to `y` is a constant. You can change the value that `y` points to, but you cannot change the value of `y`.

```
int * const y;
```

In the following example, the value that `y` points to is a constant integer and cannot be changed. However, you can change the content of `y`.

```
const int * y;
```

For other types of `volatile` and `const` variables, the position of the keyword within the definition (or declaration) is less important. For example:

```
volatile struct omega {
                         int limit;
                         char code;
                      } group;
```

provides the same storage as:

```
struct omega {
                int limit;
                char code;
             } volatile group;
```

In both examples, only the structure variable `group` receives the `volatile` qualifier. Similarly, if you specified the `const` keyword instead of `volatile`, only the structure variable `group` receives the `const` qualifier. The `const` and `volatile` qualifiers when applied to a structure or union also apply to the members of the structure or union.

Although enumeration, structure, and union variables can receive the `volatile` or `const` qualifier, enumeration, structure, and union tags do not carry the `volatile` or `const` qualifier. For example, the `blue` structure does not carry the `volatile` qualifier:

```
volatile struct whale {
                           int weight;
                           char name[8];
                      } killer;
struct whale blue;
```

The keyword `volatile` or `const` cannot separate the keywords `enum`, `struct`, and `union` from their tags.

You cannot declare or define a `volatile` or `const` function but you can define or declare a function that returns a pointer to a `volatile` or `const` object.

You can place more than one qualifier on a declaration but you cannot specify the same qualifier more than once on a declaration.

These type qualifiers are only meaningful on expressions that are lvalues.

_Packed Object Qualifier

The `_Packed` qualifier removes padding between members of structures and unions, whenever possible. However, the storage saved using packed structures and unions may come at the expense of run-time performance. Most machines access data more efficiently if it is aligned on appropriate boundaries. With packed structures and unions, members are generally not aligned on natural boundaries, and the result is that member-accessing operations (using the **.** and -> operators) are slower.

`_Packed` can only be used with structs or unions. If you use `_Packed` with other types, an error message is generated and the qualifier has no effect on the declarator it qualifies. Packed and nonpacked structures and unions have different storage layouts. However, a packed structure or union can be assigned to a nonpacked structure or union of the same type, and nonpacked structure or union can be assigned to a packed structure or union. Comparisons between packed and nonpacked structures or unions of the same type are prohibited.

If you specify the `_Packed` qualifier on a structure or union that contains a structure or union as a member, the qualifier is not passed on to the contained structure or union.

Examples

The following example uses a `_Packed` qualifier to create a packed structure:

```
struct s1
    {
    char c1;
    int i1;
    } u;               /* NOT packed      */

struct s1 v;           /* NOT packed      */
_Packed struct s1 w;   /* packed          */

_Packed struct s2
    {
    char c2;
    int i2;
    } a;               /* packed          */

struct s2 b;           /* NOT packed !!!  */
_Packed struct s2 c;   /* packed          */
```

The following table describes some declarators:

Table 7. Example Declarators

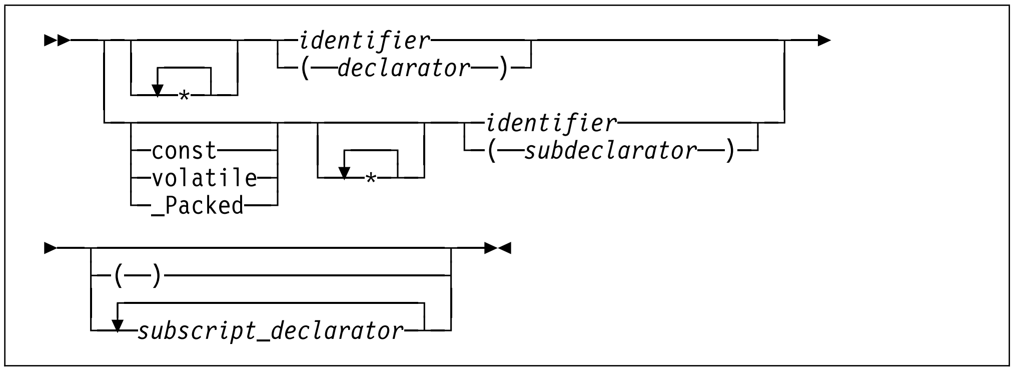
Example	Description
`int owner`	`owner` is an `int` data object.
`int *node`	`node` is a pointer to an `int` data object.
`int names[126]`	`names` is an array of 126 `int` elements.
`int *action( )`	`action()` is a function returning a pointer to an `int`.
`volatile int min`	`min` is an `int` that has the `volatile` qualifier.
`int * volatile volume`	`volume` is a `volatile` pointer to an `int`.
`volatile int * next`	`next` is a pointer to a `volatile int`.
`volatile int * sequence[5]`	`sequence` is an array of five pointers to `volatile int` objects.
`extern const volatile int op_system_clock`	`op_system_clock` is a constant and volatile integer with static storage duration and external linkage.
`_Packed struct struct_type s`	`s` is a packed structure of type `struct_type`.

Related Information

- “Arrays” on page 71
- “Enumerations” on page 63
- “Pointers” on page 78
- “Structures” on page 49
- “Unions” on page 59

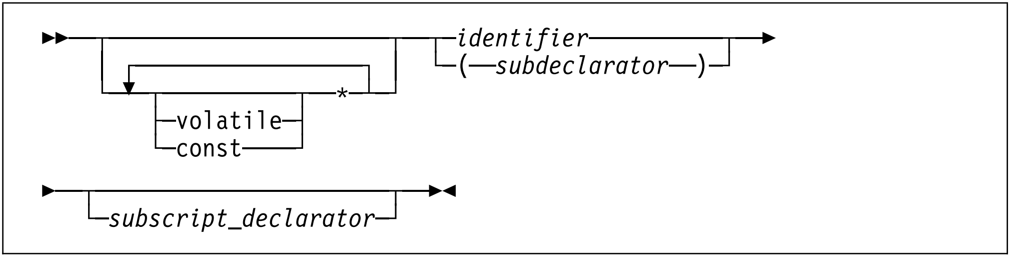
Declarators

A *declarator* designates a data object or function. Declarators appear in all data definitions and declarations and in some type definitions. A declarator has the form:

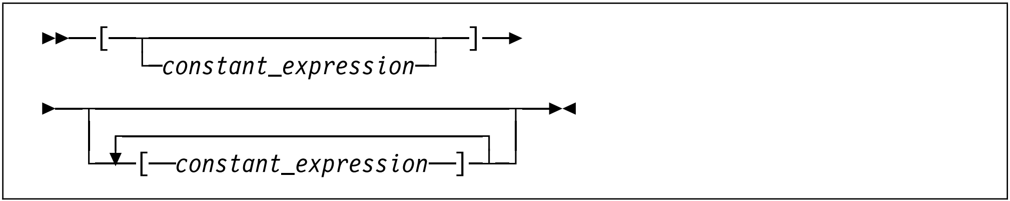


You cannot declare or define a `volatile` or `const` function.

A declarator can contain a *subdeclarator*. A subdeclarator has the form:



A subscript declarator describes the number of dimensions in an array and the number of elements in each dimension. A subscript declarator has the form:



A simple declarator consists of an identifier, which names a data object. For example, the following block scope data declaration uses `initial` as the declarator:

```
auto char initial;
```

The data object `initial` has the storage class `auto` and the data type `char`.

You can define or declare an aggregate by using a declarator that contains an identifier, which names the data object, and some combination of symbols and identifiers, which describes the type of data that the object represents. An aggregate is a structure, union, or array.

The following declaration uses `compute[5]` as the declarator:

```
extern long int compute[5];
```

Table 7 on page 69 provides more examples of declarators.

Related Information

- "volatile and const Type Qualifiers" on page 67
- "_Packed Object Qualifier" on page 68
- "Declarations and Definitions" on page 31
- "Arrays"
- "Enumerations" on page 63
- "Pointers" on page 78
- "Structures" on page 49
- "Unions" on page 59
- "Fixed-Point Decimal Data Types" on page 46

Arrays

An *array* is an ordered group of data objects. Each data object is called an *element*. All elements within an array have the same data type. An array element is placed on its natural boundary. You can pad the data where the array is a structure or a union.

Type Specifiers of Arrays

You can use any type specifier in an array declaration. Thus, array elements can be of any data type, except function. An array type specifier can be an arithmetic type, a pointer, a complete structure or union type, or another array.

Declarators of Arrays

The declarator contains an identifier followed by a *subscript declarator*. The identifier can be preceded by an asterisk (`*`), making the variable an array of pointers.

The subscript declarator describes the number of dimensions in the array and the number of elements in each dimension. A subscript declarator has the form:

```
►►──[──┬─────────────────────┬──]──►
       └─constant_expression─┘

►──┬──────────────────────────────┬──►◄
   │ ┌──────────────────────────┐ │
   └─▼─[──constant_expression──]─┴─┘
```

Each bracketed expression describes a different dimension. The constant expression must have an integral value. The value of the constant expression determines the number of elements in that dimension. The following example defines a one-dimensional array that contains four elements having type `char`:

```
char list[4];
```

The first subscript of each dimension is `0`. Thus, the array `list` contains the elements:

```
list[0]
list[1]
list[2]
list[3]
```

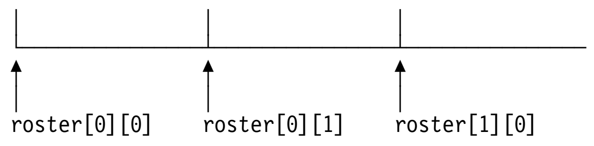
The following example defines a two-dimensional array that contains six elements of type `int`:

```
int roster[3][2];
```

Multidimensional arrays are stored in row-major order; when elements are referenced in order of increasing storage location, the last subscript varies the fastest. For example, the elements of array `roster` are stored in the order:

```
roster[0][0]
roster[0][1]
roster[1][0]
roster[1][1]
roster[2][0]
roster[2][1]
```

In storage, the elements of `roster` would be stored as:



You can leave the first (and only the first) set of subscript brackets empty in:

- Array definitions that contain initializations
- `extern` declarations
- Parameter declarations

In array definitions that leave the first set of subscript brackets empty, the compiler uses the initializer to determine the number of elements in the first dimension. In a one-dimensional array, the number of initialized elements becomes the total number of elements. In a multidimensional array, the compiler compares the initializer to the subscript declarator to determine the number of elements in the first dimension.

Initializers of Arrays

The initializer contains the `=` symbol followed by a brace-enclosed comma-separated list of constant expressions. You do not need to initialize all elements in an array. Elements that are not initialized (in `extern` and `static` definitions only) receive the value `0`.

The following definition shows a completely initialized one-dimensional array:

```
static int number[3] = { 5, 7, 2 };
```

The array `number` contains the following values:

Element	Value
`number[0]`	5
`number[1]`	7
`number[2]`	2

The following definition shows a partially initialized one-dimensional array:

```
static int number1[3] = { 5, 7 };
```

The values of `number1` are:

Element	Value
`number1[0]`	5
`number1[1]`	7
`number1[2]`	0

The following one-dimensional array definition shows the number of initialized elements in the array defining the number of elements in the array subscript.

```
static int item[ ] = { 1, 2, 3, 4, 5 };
```

The compiler gives `item` the five initialized elements:

Element	Value
`item[0]`	1
`item[1]`	2
`item[2]`	3
`item[3]`	4
`item[4]`	5

You can initialize a one-dimensional character array by specifying:

- A brace-enclosed comma-separated list of constants, each of which can be contained in a character
- A string constant (braces surrounding the constant are optional)

If you specify a string constant, the null character (`\0`) is placed at the end of the string if there is room or if the array dimensions are not specified.

The following definitions show character array initializations:

```
static char name1[ ] = { 'J', 'a', 'n' };
static char name2[ ] = { "Jan" };
static char name3[4] = "Jan";
```

These definitions create the following elements:

Element	Value	Element	Value	Element	Value
`name1[0]`	`J`	`name2[0]`	`J`	`name3[0]`	`J`
`name1[1]`	`a`	`name2[1]`	`a`	`name3[1]`	`a`
`name1[2]`	`n`	`name2[2]`	`n`	`name3[2]`	`n`
		`name2[3]`	`\0`	`name3[3]`	`\0`

Note that the following definition would result in the loss of the null character:

```
static char name[3]="Jan";
```

You can initialize a multidimensional array by:

- Listing the values of all elements you want to initialize, in the order that the compiler assigns the values. The compiler assigns values by increasing the subscript of the last dimension fastest. This form of a multidimensional array initialization looks like a one-dimensional array initialization. The following definition completely initializes the array `month_days`:

  ```
  static month_days[2][12] =
  {
   31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31,
   31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
  };
  ```

- Using braces to group the values of the elements you want initialized. You can place braces around each element or around any nesting level of elements. The following definition contains two elements in the first dimension. (You can consider these elements as rows.) The initialization contains braces around each of these two elements:

  ```
  static int month_days[2][12] =
  {
   { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
   { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
  };
  ```

You can use nested braces to initialize dimensions and elements in a dimension selectively.

The following definition explicitly initializes six elements in a 12-element array:

```
static int matrix[3][4] =
   {
     {1, 2},
     {3, 4},
     {5, 6}
   };
```

The initial values of `matrix` are:

Element	Value	Element	Value
`matrix[0][0]`	1	`matrix[1][2]`	0
`matrix[0][1]`	2	`matrix[1][3]`	0
`matrix[0][2]`	0	`matrix[2][0]`	5
`matrix[0][3]`	0	`matrix[2][1]`	6
`matrix[1][0]`	3	`matrix[2][2]`	0
`matrix[1][1]`	4	`matrix[2][3]`	0

Note: When using braces, you should place braces around each dimension (fully braced), or only use one set of braces to enclose the entire set of initializers (unbraced). Avoid putting braces around some elements and not others. An unsubscripted array name (for example, `region` instead of `region[4]`) represents a pointer whose value is the address of the first element of the array. For more information, see “Primary Expression” on page 97.

Whenever an array is used in a context (such as a parameter) where it cannot be used as an array, the identifier is treated as a pointer. The two exceptions are when an array is used as an operand of the `sizeof` or as the address (&) operator.

You cannot have more initializers than the number of elements in the array.

The following example shows a parameter declaration for a one-dimensional array:

```
test(int y[ ])
{
⋮
}
```

Examples

The following program defines a floating-point array called `prices`.

The first `for` statement prints the values of the elements of `prices`. The second `for` statement adds 5% to the value of each element of `prices`, and assigns the result to `total`, and prints the value of `total`.

EDCXRAAO

```
 /* Example of a one-dimensional array */

#include <stdio.h>
#define  ARR_SIZE  5

int main(void)
{
  static float const prices[ARR_SIZE] = { 1.41, 1.50, 3.75, 5.00, .86 };
  auto float total;
  int i;

  for (i = 0; i < ARR_SIZE; i++)
  {
    printf("price = $%.2f\n", prices[i]);
  }

  printf("\n");

  for (i = 0; i < ARR_SIZE; i++)
  {
    total = prices[i] * 1.05;

    printf("total = $%.2f\n", total);
  }
}
```

This example produces the following output:

```
price = $1.41
price = $1.50
price = $3.75
price = $5.00
price = $0.86

total = $1.48
total = $1.57
total = $3.94
total = $5.25
total = $0.90
```

The following program defines the multidimensional array `salary_tbl`. A `for` loop prints the values of `salary_tbl`.

EDCXRAAP

```
 /* Example of a multidimensional array */

#include <stdio.h>
#define  NUM_ROW     3
#define  NUM_COLUMN  5

int main(void)
{
  static int salary_tbl[NUM_ROW][NUM_COLUMN] =
  {
    {  500,  550,  600,  650,  700   },
    {  600,  670,  740,  810,  880   },
    {  740,  840,  940, 1040, 1140   }
  };
  int grade , step;

  for (grade = 0; grade < NUM_ROW; ++grade)
   for (step = 0; step < NUM_COLUMN; ++step)
   {
     printf("salary_tbl[%d] [%d] = %d\n", grade, step,
                                    salary_tbl[grade] [step]);
   }
}
```

This example produces the following output:

```
salary_tbl[0] [0] = 500
salary_tbl[0] [1] = 550
salary_tbl[0] [2] = 600
salary_tbl[0] [3] = 650
salary_tbl[0] [4] = 700
salary_tbl[1] [0] = 600
salary_tbl[1] [1] = 670
salary_tbl[1] [2] = 740
salary_tbl[1] [3] = 810
salary_tbl[1] [4] = 880
salary_tbl[2] [0] = 740
salary_tbl[2] [1] = 840
salary_tbl[2] [2] = 940
salary_tbl[2] [3] = 1040
salary_tbl[2] [4] = 1140
```

Related Information

- “Pointers” on page 78
- “Array Element Specification (Array Subscript) [ ]” on page 101
- “String Constants” on page 28
- “Declarators” on page 70
- “Initializers” on page 91
- “Conversions” on page 117

Pointers

A *pointer* type variable holds the address of a data object or a function. A pointer can refer to an object of any one data type but cannot point to an object having the `register` storage class specifier or to a bit field. Some common uses for pointers are:

- To pass the address of a variable to a function. By referencing a variable through its address, a function can change the contents of that variable. See "Calling Functions and Passing Arguments" on page 98.
- To access dynamic data structures such as linked lists, trees, and queues.
- To access elements of an array or members of a structure.
- To access an array of characters as a string.

Declarators of Pointers

The following example declares `pcoat` as a pointer to an object having type `long`:

```
extern long *pcoat;
```

If the keyword `volatile` appears before the `*`, the declarator describes a pointer to a `volatile` object. If the keyword `volatile` comes between the `*` and the identifier, the declarator describes a `volatile` pointer.

The keyword `const` operates in the same manner as the `volatile` keyword described in the preceding paragraph. In the following example, `pvolt` is a constant pointer to an object having type `short`:

```
short * const pvolt;
```

The following example declares `pnut` as a pointer to an `int` object having the `volatile` qualifier:

```
extern int volatile *pnut;
```

The following example defines `psoup` as a `volatile` pointer to an object having type `float`:

```
float * volatile psoup;
```

The following example defines `pfowl` as a pointer to an enumeration object of type `bird`:

```
enum bird *pfowl;
```

The following example declares `x` as a pointer to a function that returns a `char` object:

```
char (*x)(void);
```

Initializers of Pointers

When you use pointers in an assignment operation, you must ensure that the types of the pointers in the operation are compatible.

The following example shows compatible declarations for the assignment operation:

```
float subtotal;
float * sub_ptr;
⋮
sub_ptr = &subtotal;
printf("The subtotal is %f\n", *sub_ptr);
```

The next example shows incompatible declarations for the assignment operation:

```
double league;
int * minor;
⋮
minor = &league;     /* error */
```

The initializer is an `=` (equal sign) followed by the expression that represents the address that the pointer is to contain. The following example defines the variables `total` and `speed` as having type `double` and `amount` as having type pointer to a `double`. The pointer `amount` is initialized to point to `total`:

```
double total, speed, *amount = &total;
```

The compiler converts an unsubscripted array name to a pointer to the first element in the array. You can assign the address of the first element of an array to a pointer by specifying the name of the array. The following two sets of definitions are equivalent. Both define the pointer `student` and initialize `student` to the address of the first element in `class`:

```
int class[80];
int *student = class;
```

is equivalent to:

```
int class[80];
int *student = &class[0];
```

You can assign the address of the first character in a string constant to a pointer by specifying the string constant in the initializer. The following example defines the pointer variable `string` and the string constant `"abcd"`. The pointer `string` is initialized to point to the character `a` in the string `"abcd"`.

```
char *string = "abcd";
```

The following example defines `weekdays` as an array of pointers to string constants. Each element points to a different string. The object `weekdays[2]`, for example, points to the string `"Tuesday"`.

```
static char *weekdays[ ] =
            {
              "Sunday", "Monday", "Tuesday", "Wednesday",
              "Thursday", "Friday", "Saturday"
            };
```

A pointer can also be initialized to the integer constant `0`. Such a pointer is a *NULL pointer* that does not point to any object.

Restrictions

C/VSE supports only the pointers that are obtained in one of the following ways:

- Directly from a `malloc`, `calloc`, or `realloc` call
- As an address of a data type (that is, `&variable`)
- From constants
- Received as a parameter from another C function
- Directly from a call to an LE/VSE service that allocates storage, such as CEEGTST

Any bitwise manipulation of a pointer can result in undefined behavior.

You cannot use pointers to reference bit fields or objects having the `register` storage class specifier.

A pointer to a packed structure or union is incompatible with a pointer to a corresponding nonpacked structure or union, because packed and nonpacked objects have different memory layouts. As a result, comparisons and assignments between pointers to packed and nonpacked objects are not valid.

You can, however, perform these assignments and comparisons with type casts. Consider the following example:

```
int main(void)
{
   _Packed struct ss *ps1;
   struct ss         *ps2;
⋮
   ps1 = (_Packed struct ss *)ps2;
⋮
}
```

In the preceding example, the cast operation allows you to assign a pointer. After the assignment, `ps1` points to a packed structure and `ps2` points to an unpacked structure.

Using Pointers

Two operators that are commonly used when you are working with pointers are the `&` (address) operator and the `*` (indirection) operator. You can use the `&` operator to reference the address of an object. For example, the following statement assigns the address of `x` to the variable `p_to_x`. The variable `p_to_x` has been defined as a pointer.

```
int x, *p_to_x;

p_to_x = &x;
```

The `*` (indirection) operator enables you to access the value of the object to which a pointer refers. The following statement assigns to `y` the value of the object to which `p_to_x` points:

```
float y, *p_to_x;
⋮
y = *p_to_x;
```

The following statement assigns the value of `y` to the variable that `*p_to_x` references:

```
char y, *p_to_x,
⋮
*p_to_x = y;
```

Pointer Arithmetic

You can perform a limited number of arithmetic operations on pointers. These operations are:

- Increment and decrement
- Comparison
- Assignment

Increment and Decrement: The ++ (increment) operator increases the value of a pointer by the size of the data object to which the pointer refers. For example, if the pointer refers to the second element in an array, the ++ makes the pointer refer to the third element in the array.

The `--` (decrement) operator decreases the value of a pointer by the size of the data object to which the pointer refers. For example, if the pointer refers to the second element in an array, the `--` makes the pointer refer to the first element in the array.

If the pointer `p` points to the first element in an array, the following expression causes the pointer to point to the third element in the same array:

```
p = p + 2;
```

Addition and Subtraction: If you have two pointers that point to the same array, you can subtract one pointer from the other. This operation yields the number of elements in the array that separate the two addresses to which the pointers refer. Addition cannot be performed between two pointers. However, both addition and subtraction may be performed between a pointer and an integer.

Comparison: You can compare two pointers with the following operators: `==`, `!=`, <, >, <=, and >=. See Chapter 4, “Expressions and Operators” on page 93 for more information on these operators.

Pointer comparisons are defined only when the pointers point to elements of the same array.

Assignment: You can assign to a pointer the address of a data object, the value of another compatible pointer or the `NULL` pointer.

Examples

The following program contains pointer arrays:

EDCXRAAQ

```
 /* Example of how to use pointer arrays
    This example searches for the first occurrence of a specified
    character string in an array of character strings */

#include <stdio.h>
#include <stdlib.h>
#define  SIZE  20

int main(void)
{
   static char *names[ ] = { "Jim", "Amy", "Mark", "Sue", NULL };
   char * find_name(char **, char *);
   char new_name[SIZE], *name_pointer;

   printf("Enter name to be searched.\n");
   scanf("%s", new_name);
   name_pointer = find_name(names, new_name);
   printf("name %s%sfound\n", new_name,
          (name_pointer == NULL) ? " not " : " ");
   exit(EXIT_FAILURE);
} /* End of main */

 /* Function find_name.
    This function searches an array of names to see if a given name
    already exists in the array.  It returns a pointer to the name
    or NULL if the name is not found. */

 /* char **arry is a pointer to arrays of pointers (existing names) */
 /* char *strng is a pointer to character array entered (new name)  */

char * find_name(char **arry, char *strng)
{
   for (; *arry != NULL; arry++)         /* for each name          */
   {
      if (strcmp(*arry, strng) == 0)     /* if strings match       */
         return(*arry);                  /* found it!              */
   }
   return(*arry);                        /* return the pointer     */
} /* End of find_name */
```

Interaction with the preceding program could produce the following sessions:

Output `Enter name to be searched.`

Input `Mark`

Output `name Mark found`

or:

Output `Enter name to be searched.`

Input `Bob`

Output `name Bob not found`

Related Information

- “Address & ” on page 104
- “Indirection * ” on page 104
- “Declarators” on page 70
- “volatile and const Type Qualifiers” on page 67
- “Initializers” on page 91

Functions

Function Definition

A *function definition* specifies the name, formal parameters, and body of a function. You can also specify the function’s return type and storage class. A function definition has the form:

```
►►──┬────────┬──┬────────────────────┬──┬────────────────┬──────►
    ├─extern─┤  └─linkage_specifier──┘  └─type_specifier─┘
    └─static─┘

                         ┌──────────────────────────────┐
►──function_declarator───▼──────────────────────────────┴──────►
                         └─(──parameter_declaration──)─┘

►──block_statement──►◄
```

There are two ways to define a function: prototype and nonprototype. The prototype method is preferred because of the parameter type checking that can be performed.

A function definition (either prototype or nonprototype) contains the following:

- The optional storage class specifier `extern` or `static`, which determines the scope of the function. If a storage class specifier is not given, the function has external linkage.
- An optional type specifier, which determines the type of value that the function returns. If a type specifier is not given, the function has type `int`.
- A function declarator, which provides the function with a name, can further describe the type of the value that the function returns, and can list any parameters (and their types) that the function expects. The parameters that the function is expecting are enclosed in parentheses.
- A block statement, which contains data definitions, data declarations, and code.

In addition, the nonprototype function definition may also have parameter declarations, that describe the types of parameters that the function receives. In nonprototype functions, parameters that are not declared have type `int`.

A function can be called by itself or by other functions. Unless a function definition has the storage class specifier `static`, the function can also be called by functions that appear in other files. If the function has a storage class specifier of `static`, it can only be directly invoked from within the same source file. If a function has the storage class specifier `static` or a return type other than `int`, the function definition or a declaration for the function must appear before, and in the same file as, a call to the function. If a function definition has external linkage and a return type of `int`, calls to the function can be made before it is visible because an implicit declaration of `extern int func();` is assumed. All declarations for a given function must be compatible; that is, the return type is the same and the parameters have the same type.

The default type for the return value and parameters of a function is `int`, and the default storage class specifier is `extern`. If the function does not return a value or it is not passed any parameters, use the keyword `void` as the type specifier and use an empty parameter list.

```
void foo(void);          /* a function declaration that     */
                         /* says function foo() returns     */
                         /* nothing and takes no arguments  */
```

You can include ellipses (...) at the end of your parameter list to indicate that a variable number of arguments will be passed to the function. Parameter promotions are performed, and no type checking is done.

In the following example, the function `foo()` takes at least one integer parameter; the types and number of the other arguments are not known:

```
int foo(int a, ...)
```

You cannot declare a function as a struct or union member.

A function cannot have a return type of function, array, or any type having the `volatile` or `const` qualifier. However, it can return a pointer to an object with a `volatile` or `const` type.

You cannot define an array of functions. You can, however, define an array of pointers to functions. In the following example, `ary` is an array of two function pointers. Type casting is performed to the values assigned to `ary` for compatibility.

EDCXRAAT

```
 /* Example that uses an array of pointers to functions */

#include <stdio.h>
int func1(void);
void func2(double a);
int main(void)
{
   double num;
   int retnum;
   void (*ary[2]) ();
   ary[0] = ((void(*)())func1);
   ary[1] = ((void(*)())func2);

   retnum=((int (*)())ary[0])();      /*  calls func1  */
   printf("number returned = %i\n", retnum);
   ((void (*)(double))ary[1])(num);   /*  calls func2  */
}

int func1(void)
{
int number=3;
return number;
}

void func2(double a)
{
a=333.3333;
printf("result of func2 = %f\n", a);
}
```

The following example is a complete definition of the function `sum`:

```
int sum(int x,int y)
{
   return(x + y);
}
```

The function `sum()` has external linkage, returns an object that has type `int`, and has two parameters of type `int` declared as `x` and `y`. The function body contains a single statement that returns the sum of `x` and `y`.

Function Declarator

The *function declarator* names the function and lists the function parameters. A function declarator contains an identifier that names the function and a list of the function parameters.

There are two types of function declarators: prototype and nonprototype. In a prototype function declarator, the types of all parameters in the parameter list are specified. In a nonprototype function declarator, only the identifiers of the parameters are specified; the parameter types are not specified. Prototype function declarators are preferred because of the parameter checking that can be performed.

Prototype Function Declarator

A prototype function declarator has the form:

```
►►──identifier──(──►

►──┬──────────────────────────────────────────────────┬──)──►◄
   │ ┌─,──────────────────────────┐                   │
   └─▼─type_specifier──┬───────────┬─┴──┬────────────┬──┘
                       └─parameter─┘    └─,──...───┘
```

Declare each parameter within the function declarator for prototype function declarations. Any calls to the function must pass the same number of arguments as there are parameters in the declaration.

To indicate that a function does not receive any values, use the keyword `void` in place of the parameter. For example:

```
int stop(void)
{
}
```

The example below contains a function declarator `sort` with `table` declared as an array of `int` and `length` declared as type `int`.

Note: Arrays as parameters are implicitly converted to a pointer to the type, and therefore `table` is passed as a pointer to an `int`.

EDCXRAAU

```
 /* Example of function declarators
    Note that arrays as parameters are implicitly
    converted to a pointer to the type */

void sort(int table[ ], int length);

int main(void)
{
  int tableffl “={1,5,8,4};
  int length=4;
  printf("length is %d\n",length);
  sort(table,length);
}
```

```
void sort(int table[ ], int length)
{
  int i, j, temp;

  for (i = 0; i < length -1; i++)
    for (j = i + 1; j < length; j++)
      if (table[i] > table[j])
      {
        temp = table[i];
        table[i] = table[j];
        table[j] = temp;
      }
}
```

The following examples contain prototype function declarators:

```
double square(float x);
int area(int x,int y);
static char *search(char);
```

The example below illustrates how a `typedef` identifier can be used in a function declarator:

```
typedef struct tm_fmt { int minutes;
                      int hours;
                      char am_pm;
                      } struct_t;
long time_seconds(struct_t arrival)
   {
   arrival.minutes = 30;
   arrival.hours = 12;
   arrival.am_pm = 'p';
   }
```

The following function `set_date()` declares a pointer to a structure of type `date` as a parameter. `date_ptr` has the storage class specifier `register`.

```
set_date(register struct date *date_ptr)
{
  date_ptr->mon = 12;
  date_ptr->day = 25;
  date_ptr->year = 87;
}
```

Nonprototype Function Declarator

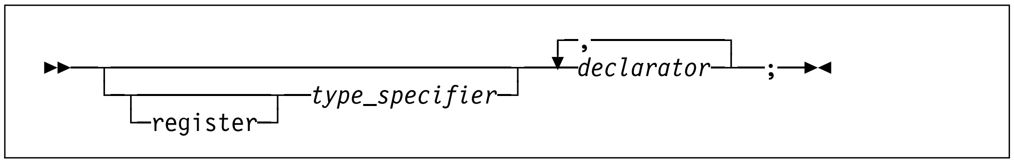
A nonprototype function declarator has the form:

```
                     ┌─,──────────┐
►►──identifier──(──▼────────────┴──)──►◄
                     └─parameter─┘
```

Each parameter should be declared in a parameter declaration list following the declarator. If a parameter is not declared, it has type `int`.

`char` and `short` parameters are widened to `int`, and `float` to `double`. There is no default argument promotion for decimal types. No type checking between the argument type and the parameter type is done for nonprototyped functions. As well, there are no checks to ensure that the number of arguments matches the number of parameters.

A parameter declaration determines the storage class specifier and the data type of the value. It has the form:



The only storage class specifier allowed is the `register` storage class specifier. Any type specifier for a parameter is allowed. If you do not specify the `register` storage class specifier, the parameter will have the `auto` storage class specifier. If you omit the type specifier and you are not using the prototype form to define the function, the parameter will have type `int`.

```
int func(i,j)
{
   /*  i and j have type int  */
}
```

You cannot declare a parameter in the parameter declaration list if it is not listed within the declarator.

The following is an example of a nonprototype function declarator followed by a parameter declaration list:

```
int func(a, b)
char a;
float b;
{
⋮
}
```

Function Body

The function body is a block statement. (For more information on block statements, see “Block” on page 127.) The following function has an empty body:

```
void stub1(void)
{
}
```

The following function body contains a definition for the integer variable `big_num`, an `if-else` control statement, and a call to the function `printf()`:

```
void largest(int num1, int num2)
{
   int big_num;

   if (num1 >= num2)
      big_num = num1;
   else
      big_num = num2;

   printf("big_num = %d\n", big_num);
}
```

Function Declarations

A function declaration establishes the name and the parameters of the function. A function is declared implicitly by its appearance in an expression if it has not been defined or declared previously; the implicit declaration is equivalent to a declaration of `extern int` *func_name*`(void)`.

If the called function returns a value that has a type other than `int`, you must declare the function before the function call. Even if a called function returns a type `int`, explicitly declaring the function prior to its call is good programming practice.

Some declarations do not have parameter lists; the declarations simply specify the types of parameters and the return values, as in the following example:

```
int func(int,long);
```

Examples

The following example defines the function `absolute` with the return type `double`. Because this is a noninteger return type, `absolute` is declared prior to the function call.

EDCXRAAV

```
 /* Example of a function defined prior to the function call
    because it returns a noninteger value */

#include <stdio.h>
double absolute(double);

int main(void)
{
   double f = -3.0;

   printf("absolute number = %f\n", absolute(f));
}

double absolute(double number)
{
   if (number < 0.0)
      number = -number;

   return (number);
}
```

Specifying a return type of `void` on a function declaration indicates that the function does not return a value. The following example defines the function `absolute` with the return type `void`. Within the function `main()`, `absolute` is declared with the return type `void`.

EDCXRAAW

```
 /* Example of using void as the return type of a function */

#include <stdio.h>

int main(void)
{
  void absolute(float);
  float f = -8.7;

  absolute(f);
}

void absolute(float number)
{
  if (number < 0.0)
    number = -number;

  printf("absolute number = %f\n", number);
}
```

Related Information

- "Declaration" on page 37

typedefs

`typedef` declarations in C allow you to define your own identifiers that can be used in place of C type specifiers such as `int`, `float`, and `double`. The data types you define using `typedef` are not new data types. The identifiers you define are synonyms for the primary data types used by the C language or data types derived by combining the primary data types.

```
►►──typedef──type_specifier──identifier──;──►◄
```

A `typedef` declaration does not reserve storage.

When an object is defined using a `typedef` identifier, the properties of the defined object are exactly the same as if the object were defined by explicitly listing the data type associated with the identifier.

Examples

The following statements declare `LENGTH` as a synonym for `int`, and then use this `typedef` to declare `length`, `width`, and `height` as integral variables:

```
typedef int LENGTH;
LENGTH length, width, height;
```

The preceding lines are equivalent to the following:

```
int length, width, height;
```

Similarly, `typedef` can be used to define a `structure` type. For example:

```
typedef struct {
                int kilos;
                int grams;
               } WEIGHT;
```

The structure `WEIGHT` can then be used in the following declarations:

```
WEIGHT  chicken, cow, horse, whale;
```

The following is an example using `typedef` to define an enumerated type:

```
enum boolean { FALSE, TRUE };
typedef enum boolean BOOL;
BOOL done = FALSE;
```

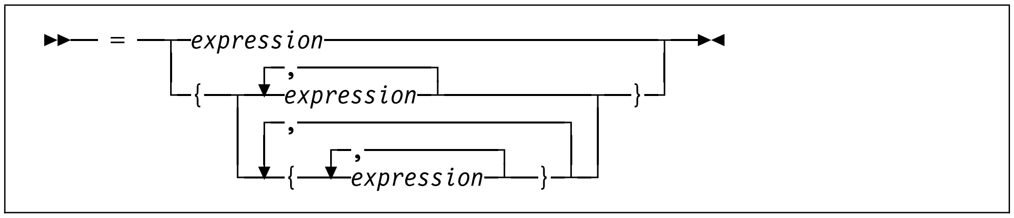
Related Information

- "Characters" on page 44
- "Floating-Point" on page 45
- "Fixed-Point Decimal Data Types" on page 46
- "Integers" on page 47
- "void Type" on page 49
- "Arrays" on page 71
- "Enumerations" on page 63
- "Pointers" on page 78
- "Structures" on page 49
- "Unions" on page 59

Initializers

An *initializer* is an optional part of a data declaration that specifies an initial value of a data object.

An initializer has the form:



The initializer consists of the `=` symbol followed by an initial *expression* or a braced list of initial expressions separated by commas. The number of initializers should not be more than the number of elements to be initialized. The initial expression evaluates to the first value of the data object.

To assign a value to a scalar object, use the simple initializer: `=` *expression*. For example, the following data definition uses the initializer `= 3` to set the initial value of `group` to `3`:

```
int group = 3;
```

For unions and structures, the set of initial expressions must be enclosed in `{ }` (braces) unless the initializer is a string literal. If the initializer of a character string

is a string literal, the `{ }` are optional. Individual expressions must be separated by commas, and groups of expressions can be enclosed in braces and separated by commas. The number of initializers must be less than or equal to the number of objects being initialized.
In the following example, only the first eight elements of the array `grid` are explicitly initialized. The remaining four elements that are not explicitly initialized are initialized as if you explicitly initialized them to zero.

```
static short grid[3] [4] = {0, 0, 0, 1, 0, 0, 1, 1};
```

The initial values of `grid` are:

Element	Value	Element	Value
grid[0] [0]	0	grid[1] [2]	1
grid[0] [1]	0	grid[1] [3]	1
grid[0] [2]	0	grid[2] [0]	0
grid[0] [3]	1	grid[2] [1]	0
grid[1] [0]	0	grid[2] [2]	0
grid[1] [1]	0	grid[2] [3]	0

Initialization considerations for each data type are described in the section for that data type.

C Data Mapping

The System/370 architecture has the following boundaries in its memory mapping:

- Byte
- Halfword
- Fullword
- Doubleword

The code produced by the C compiler places data types on natural boundaries. Some examples are:

- Byte boundary for `char`
- Halfword boundary for `short int`
- Fullword boundary for `int`
- Fullword boundary for `long int`
- Fullword boundary for *pointer to ...*
- Fullword boundary for `float`
- Doubleword boundary for `double`
- Doubleword boundary for `long double`

The C/VSE compiler places external variables in the CSECT static control section and dynamic variables in storage allocated dynamically.

Chapter 4. Expressions and Operators

This chapter describes C language expressions. The evaluation of expressions is based on the operators that the expressions contain and the context in which they are used.

Most expressions can contain several different, but related, types of operands. The following *type classes* describe related types of operands:

Integral	Character objects and constants, objects having an enumeration type, and objects having the type `short`, `int`, `long`, `signed char`, `unsigned short`, `unsigned int`, `unsigned long` or `unsigned char`.
Arithmetic	Integral objects and objects having the type `float`, `double`, `long double`, and `decimal`.
Scalar	Arithmetic objects and pointers of any type.
Aggregate	Arrays and structures.

Many operators cause conversions from one data type to another. Conversions are discussed in “Conversions” on page 117.

Grouping and Evaluating Expressions

Two operator characteristics determine how operands group with operators: *precedence* and *associativity*. Precedence provides a priority system for grouping different types of operators with their operands. Associativity provides a left-to-right or right-to-left order for grouping operands to operators that have the same precedence. For example, in the following statements, the value of `5` is assigned to both `a` and `b` because of the right-to-left associativity of the `=` operator. The value of `c` is assigned to `b` first, and then the value of `b` is assigned to `a`.

```
b = 9;
c = 5;
a = b = c;
```

When the order of expression evaluation is not specified, you can explicitly force the grouping of operands with operators by using parentheses.

In the expression `a + b * c / d`, the `*` and `/` operations are performed before `+` because of precedence. `b` is multiplied by `c` before it is divided by `d` because of associativity.

The following table lists the C language operators in order of precedence and shows the direction of associativity for each operator. The primary operators have the highest precedence. The comma operator has the lowest precedence. Operators that appear in the same group have the same precedence.

Table 8. Operator Precedence and Associativity

Operator Name	Associativity	Operators
Primary	Left to right	`()` `[ ]` `.` `->`

Unary	Right to left	`++` `--` `-` `+` `!`
		`~` `&` `*`
		(*typename*)
		`sizeof` `digitsof` `precisionof`

Multiplicative	Left to right	`*` `/` `%`
Additive	Left to right	`+` `-`
Bitwise shift	Left to right	`<<` `>>`
Relational	Left to right	`<` `>` `<=` `>=`
Equality	Left to right	`==` `!=`
Bitwise logical AND	Left to right	`&`
Bitwise exclusive OR*	Left to right	`^`
Bitwise inclusive OR*	Left to right	|
Logical AND	Left to right	`&&`
Logical OR*	Left to right	||
Conditional	Right to left	`? :`

Assignment*	Right to left	`=` `+=` `-=` `*=`
		`/=` `<<=` `>>=` `%=`
		`&=` `^=` |=

Comma	Left to right	`,`

Note: * These are described in “Character Set” on page 7.

Refer to the *LE/VSE C Run-Time Programming Guide* for more information on the locale of the ¬ or ^ operator, the | or ¦ operator, and the || or ¦¦ operator.

The order of evaluation for function call arguments or for the operands of binary operators is not specified. Avoid writing such ambiguous expressions as:

```
z = (x * ++y) / func1(y);
func2(++i, x[i]);
```

In the example above, `++y` and `func1(y)` may not be evaluated in the same order by all C language implementations. If `y` has the value of 1 before the first statement, it is not known whether the value of 1 or 2 is passed to `func1()`. In the second statement, if `i` had the value of 1, it is not known whether the first or second array element of `x[ ]` is passed as the second argument to `func2()`.

The order of grouping operands with operators in an expression containing more than one instance of an operator with both associative and commutative properties is not specified. The operators that have the same associative and commutative properties are: `*`, `+`, `&`, ¦ (or |), and ^ (or ¬). The grouping of operands can be forced by grouping the expression in parentheses. For more information on the locale of these operators, refer to the *LE/VSE C Run-Time Programming Guide*.

Examples

The parentheses in the following expressions explicitly show how the C language groups operands and operators. If parentheses did not appear in these expressions, the operands and operators would be grouped in the same manner as indicated by the parentheses.

```
total = (4 + (5 * 3));
total = (((8 * 5) / 10) / 3);
total = (10 + (5/3));
```

The following expression contains operators that are both associative and commutative:

```
total = price + prov_tax + city_tax;
```

The C language does not specify the order of evaluation of operands, and this may lead to unexpected results. In the following expression, each function call may be modifying the same global variables. These side effects may result in different values for the expression depending on the order in which the functions are called:

```
a = b() + c() + d();
```

If the expression contains operators that are both associative and commutative and the order of grouping operands with operators can affect the result of the expression, separate the expression into several expressions. For example, the following expressions could replace the previous expression if the called functions do not produce any side effects that affect the variable `a`.

```
a = b();
a += c();
a += d();
```

Related Information

- "Parenthesized Expression ( )" on page 97

Lvalue

An *lvalue* is an expression that represents an object. A *modifiable lvalue* is an expression representing an object that can be changed. A modifiable lvalue is the left operand in an assignment expression. The left operand must be an lvalue.

Usage

All assignment operators evaluate their right operand and assign that value to their left operand. The left operand must evaluate to a reference to an object.

The assignment operators are not the only operators that require an operand to be an lvalue. The address operator requires an lvalue as an operand, while the increment and the decrement operators require a modifiable lvalue as an operand.

Examples

Expression	Lvalue
`x = 42;`	`x`
`*ptr = newvalue;`	`*ptr`
`a++`	`a`

Related Information

- “Assignment Expression” on page 114
- “Address &” on page 104
- “Structure and Union Member Specification . –>” on page 101

Constant Expression

A *constant expression* is an expression with a value that can be evaluated during compilation rather than at run time. It evaluates to a constant value representable by its type.
A constant expression can be:

- Integral constant expression
- Arithmetic constant expression
- Address constant expression

An integral constant expression has integral type. It is composed of integer constants, character constants, enumeration constants, sizeof expressions, and floating-point constants that are operands of casts. Cast operators only convert arithmetic types to integral types (except in sizeof).

An integral constant expression is required:

- In a bit-field width specifier
- In the subscript declarator, as the description of an array bound
- In an enumerator, as the numeric value of an enumeration constant
- In the value of a case constant
- In the `#if` preprocessing directive

An arithmetic constant expression has arithmetic type. It is composed of integer constants, character constants, enumeration constants, `sizeof` expressions, and floating-point constants. Cast operators only convert arithmetic types to arithmetic types (except in `sizeof` expressions).

An address constant expression is a pointer to an object of static storage duration or to a function designator.

A constant expression in an initializer can be:

- An arithmetic constant
- A null pointer constant
- An address constant
- An address constant for an object (plus or minus an integral constant)

In a file scope data definition, the initializer must evaluate to a constant or to the address of a static storage (`extern` or `static`) object (plus or minus an integer constant) that is defined or declared earlier in the file. Thus, the constant expression in the initializer can contain integer, character, enumeration, and float constants, casts to any type, `sizeof` expressions, and addresses (possibly modified by constants) of static objects.

In `#if`, constant expressions:

- Are integral constant expressions with no cast operator
- Can contain defined identifier or defined (identifier) expressions

Examples

The following examples show constants used in expressions.

Expression	Constant
`x = 42;`	`42`
`extern int cost = 1000;`	`1000`

Related Information

- “Arrays” on page 71
- “File Scope Data Declarations” on page 32
- “switch” on page 140
- “Enumerations” on page 63
- “Structures” on page 49
- “Conditional Compilation” on page 157

Primary Expression

A *primary expression* can be:

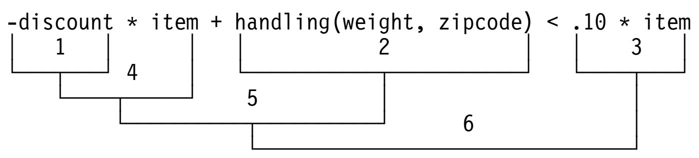
- An identifier
- A string literal
- A parenthesized expression
- A constant expression
- A function call
- An array element specification
- A structure or union member specification

All primary operators have the same precedence and have left-to-right associativity.

Parenthesized Expression ( )

You can use parentheses to explicitly force the order of expression evaluation. The following expression does not contain any parentheses used for grouping operands and operators. The parentheses surrounding `weight, zipcode` are used to form a function call. Notice how the operands and operators are grouped in this expression according to the rules for operator precedence and associativity.

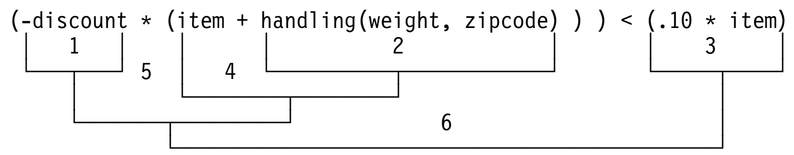
This expression is evaluated in the order shown:



```
-discount * item + handling(weight, zipcode) < .10 * item
```

The following expression is similar to the previous expression. This expression, however, contains parentheses that change how the operands and operators group.

This expression is evaluated in the order shown:



```
(-discount * (item + handling(weight, zipcode) ) ) < (.10 * item)
```

Function Call ( )

A *function call* is a primary expression containing a parenthesized argument list. The argument list can contain any number of expressions separated by commas. For example:

```
stub()
overdue(account, date, amount)
notify(name, date + 5)
report(error, time, date, ++num)
```

The arguments are evaluated, and each parameter is assigned the value of the corresponding argument. Assigning a value to a parameter within the function body changes the value of the parameter within the function, but has no effect on the argument.

Calling Functions and Passing Arguments

A function call specifies a function name and a list of arguments. The calling function passes the value of each argument to the specified function. The argument list is enclosed by parentheses, and each argument is separated by a comma. The argument list can be empty.

The arguments to a function are evaluated before the function is called. When an argument is passed in a function call, the function receives a copy of the argument value. If the value of the argument is an address, the called function can use indirection to change the contents pointed to by the address. If a function or array is passed as an argument, the argument is converted to a pointer that points to the function or type of array element.

Arguments passed to parameters in prototype declarations are converted to the declared parameter type. For nonprototype function declarations, `char` and `short` parameters are promoted to `int`, and `float` to `double`. There is no default argument promotion for decimal types.

You can pass a packed structure argument to a function expecting a nonpacked structure of the same type and vice versa. (The same applies to packed and nonpacked unions.)

Note: If you do not use a function prototype and you send a packed structure when a nonpacked structure is expected, a run-time error may occur.

The order in which arguments are evaluated and passed to the function is implementation defined. For example, the following sequence of statements calls the function `tester()`:

```
int x;
x = 1;
tester(x++, x);
```

The call to `tester()` in the preceding example may produce different results on different compilers. Depending on the implementation, `x++` may be evaluated first, or `x` may be evaluated first. To avoid ambiguity, if you want `x++` to be evaluated first, you can replace the preceding sequence of statements with the following:

```
int x, y;
x = 1;
y = x++;
tester(y, x);
```

Examples

The following statement calls the function `startup` and passes no parameters:

```
startup();
```

The following function call causes copies of `a` and `b` to be stored in a local area for the function `sum()`. The function `sum()` is executed using the copies of `a` and `b`.

```
total = sum(a, b);
```

The following function call passes the value `2` and the value of the expression `a + b` to `sum()`:

```
total = sum(2, a + b);
```

The following statement calls the functions `printf()` and `sum()`. `sum()` receives the values of `a` and `b`. `printf()` receives a character string and the return value of the function `sum()`:

```
printf("sum = %d\n", sum(a,b));
```

The following program passes the value of `count` to the function `increment()`. `increment()` increases the value of the parameter `x` by `1`.

EDCXRAAX

```
 /* Example of how to pass a parameter to a function */

#include <stdio.h>
void increment(int);

int main(void)
{
  int count = 5;

  /* value of count is passed to the function */
  increment(count);
  printf("count = %d\n", count);
}

void increment(int x)
{
  ++x;
  printf("x = %d\n", x);
}
```

The output illustrates that the value of `count` in `main()` remains unchanged:

```
x = 6
count = 5
```

In the following program, `main()` passes the address of `count` to `increment()`. The function `increment()` was changed to handle the pointer. The parameter `x` is declared a pointer. The contents to which `x` points are then incremented.

EDCXRAAY

```
 /* Example of how to pass an address to a function */

#include <stdio.h>

int main(void)
{
  void increment(int *x);
  int count = 5;

  /* address of count is passed to the function */
  increment(&count);
  printf("count = %d\n", count);
}

void increment(int *x)
{
  ++*x;
  printf("*x = %d\n", *x);
}
```

The output shows that the variable `count` is increased. The following is an example of a pass by reference:

```
*x = 6
count = 6
```

Array Element Specification (Array Subscript) [ ]

A primary expression followed by an expression in `[ ]` (square brackets) specifies an element of an array. The expression within the square brackets is referred to as a subscript.

The primary expression must have a pointer type, and the subscript must have integral type. The result of an array subscript is an lvalue.

The first element of each array has the subscript `0`. Thus, the expression `contract[35]` refers to the 36th element in the array `contract`.

In a multidimensional array, you can reference each element (in the order of increasing storage locations) by incrementing the rightmost subscript. For example, the following statement gives the value `100` to each element in the array `code[4][3][6]`:

```
for (first = 0; first <= 3; ++first)
   for (second = 0; second <= 2; ++second)
      for (third = 0; third <= 5; ++third)
         code[first][second][third] = 100;
```

Related Information

- “Arrays” on page 71

Structure and Union Member Specification . ->

Two primary operators enable you to specify structure and union members: **.** (dot) and `->` (arrow).

The dot (a period) and arrow (formed by a minus and a greater than symbol) operators are always preceded by a primary expression and followed by an identifier.

When you use the dot operator, the primary expression must be an instance of a type of structure or union, and the identifier must name a member of that structure or union. The value of the entire expression is the value associated with the named structure or union member. The result is an lvalue if the first expression is an lvalue.

Some example dot expressions:

```
roster[num].name
roster[num].name[1]
```

When you use the arrow operator, the primary expression must be a pointer to a structure or a union, and the identifier must name a member of the structure or union. The value of the entire expression (which is an lvalue) is the value of the named structure or union member to which the pointer expression refers; it is also an lvalue.

```
roster -> name
```

Related Information

- “Unions” on page 59
- “Structures” on page 49

Unary Expression

A *unary expression* contains one operand and a unary operator. All unary operators have the same precedence and have right-to-left associativity.

Increment ++

The ++ (increment) operator adds 1 to the value of the operand, or if the operand is a pointer, increments the operand by the size of the object to which it points. Since the operator changes the value of the operand, the operand must be a modifiable lvalue.

You can place the ++ before or after the operand. If it appears before the operand, the operand is incremented; then the incremented value is the value in the expression. If you place the ++ after the operand, the current value of the operand is used in the expression. Then the operand is incremented. For example:

```
play = ++play1 + play2++;
```

is equivalent to the following three expressions:

```
play1 = play1 + 1;
play  = play1 + play2;
play2 = play2 + 1;
```

The result has the same type as the operand after integral promotion, but is not an lvalue.

Related Information

- “Usual Arithmetic Conversions” on page 117

Decrement --

The -- (decrement) operator subtracts 1 from the value of the operand, or if the operand is a pointer, decreases the operand by the size of the object to which it points. Since the operator changes the value of the operand, the operand must be a modifiable lvalue.

You can place the -- before or after the operand. If it appears before the operand, the operand is decremented, and the decremented value is used in the expression. If the -- appears after the operand, the current value of the operand is used in the expression and the operand is decremented.

For example:

```
play = --play1 + play2--;
```

is equivalent to the following three expressions:

```
play1 = play1 - 1;
play = play1 + play2;
play2 = play2 - 1;
```

The result has the same type as the operand after integral promotion, but is not an lvalue.

Related Information

- “Usual Arithmetic Conversions” on page 117

Unary Plus +

The + (unary plus) operator maintains the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

The result of the unary plus expression has the same type as the operand after any integral promotions (for example, `char` to `int`).

Related Information

- “Usual Arithmetic Conversions” on page 117

Unary Minus –

The - (unary minus) operator negates the value of the operand. The operand can have any arithmetic type. The result is not an lvalue.

For example, if `quality` has the value `100`, `-quality` has the value `-100`.

The result of the unary minus expression has the same type as the operand after any integral promotions (for example, `char` to `int`).

Related Information

- “Usual Arithmetic Conversions” on page 117

Logical Negation !

The ! (logical negation) operator determines whether the operand evaluates to `0` (false) or nonzero (true). The expression yields the value `1` (true) if the operand evaluates to `0`, and yields the value `0` (false) if the operand evaluates to a nonzero value. The operand must have a scalar data type, but the result of the operation is always type `int` and is not an lvalue.

The following two expressions are equivalent:

```
!right;
right == 0;
```

Related Information

- “Usual Arithmetic Conversions” on page 117

Bitwise Negation ˜

The ˜ (bitwise negation) operator yields the bitwise complement of the operand. In the binary representation of the result, every bit has the opposite value of the same bit in the binary representation of the operand. The operand must be an integral type. The result has the same type as the operand after integral promotion, but is not an lvalue.

Let `x` represent the value `5`. The 16-bit binary representation of `x` is:

```
0000000000000101
```

The expression `¬x` yields the following result (represented here as a 16-bit binary number):

```
1111111111111010
```

The 16-bit binary representation of `¬0` is:

```
1111111111111111
```

Related Information

- “Usual Arithmetic Conversions” on page 117

Address &

The `&` (address) operator yields a pointer to its operand. The operand must be an lvalue or function designator. It cannot be a bit field, nor can it have the storage class specifier `register`. The result is a pointer to the type of the operand. Thus, if the operand has type `int`, the result is a pointer to an object having type `int`. The result is not an lvalue.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the following expression assigns the address of the variable `y` to the pointer `p_to_y`:

```
p_to_y = &y;
```

Related Information

- “Pointers” on page 78

Indirection *

The `*` (indirection) operator determines the object referred to by the pointer operand. The operand cannot be a pointer to void. The operation yields an lvalue or a function designator if the operand points to a function. If the operand is a pointer to *type*, the result has type *type*. Do not apply the indirection operator to any pointer that contains an address that is not valid, such as `NULL`.

If `p_to_y` is defined as a pointer to an `int` and `y` as an `int`, the expressions:

```
p_to_y = &y;
*p_to_y = 3;
```

cause the variable `y` to receive the value `3`.

Related Information

- “Pointers” on page 78

Cast

A *cast* operator converts the type of the operand to a specified data type and performs the necessary conversions to the operand for the type. The cast operator is a parenthesized type specifier or an expression. This type and the operand must be scalar; the type may also be `void`. The result has the type of the specified data type but is not an lvalue. For decimal operations, the cast operator suppresses error messages and run-time exceptions or both.

The following expression contains a cast expression to convert an operand of type `int` to a value of type `double`:

```
int x;
printf("x=%f\n", (double)x);
```

The function `printf()` receives the value of `x` as a `double`. The variable `x` remains unchanged by the cast.

Related Information

- "Conversions" on page 117

Size of an Object

The `sizeof` operator yields the size in *bytes* of the operand. The `sizeof` operation cannot be performed on a bit field, a function, or an incomplete type such as `void`. The operand may be the parenthesized name of a type or expression. The compiler must be able to evaluate the size at compile time. The expression is not evaluated; there are no side effects. For example, the value of `b` is `5` from initialization to program termination:

```
#include <stdio.h>

int main(void){
  int b = 5;
  sizeof(b++);
}
```

The result is an integer constant.

The size of a `char` object is the size of a byte. Given that the variable `x` has type `char`, the expression `sizeof(x)` always evaluates to `1`.

The result of a sizeof operation has type `size_t`. `size_t` is an unsigned integral type defined in the `<stddef.h>` header.

The compiler determines the size of an object on the basis of its definition. The `sizeof` operator does not perform any conversions. However, if the operand contains operators that perform conversions, the compiler takes these conversions into consideration. The following expression causes the usual arithmetic conversions to be performed. The result of the expression `x + 1` has type `int` (if `x` has type `char`, `short`, or `int` or any enumeration type) and is equivalent to `sizeof(int)`:

```
sizeof (x + 1)
```

When you perform the `sizeof` operation on an array, the result is the total number of bytes in the array. The compiler does not convert the array to a pointer before evaluating the expression.

You can use a `sizeof` expression wherever an integral constant is required. One of the most common uses for the `sizeof` operator is to determine the size of objects that are being communicated to or from storage allocation, input, and output functions.

For portability of code, you should use the `sizeof` operator to determine the size that a data type represents. In this instance, the name of the data type must be placed in parentheses after the `sizeof` operator. For example:

```
sizeof(int)
```

When you use the `sizeof` operator with `decimal(`*n*`,`*p*`)`, the result is the total number of bytes occupied by the decimal type. In C/VSE, decimal data types are implemented using the native packed decimal format. Each digit occupies half a byte. The sign occupies an additional half byte. For example:

```
sizeof(decimal(10,2))
```

will give you a result of 6 bytes.

digitsof and precisionof

The `digitsof` and `precisionof` operators yield information about `decimal` types or an expression of the `decimal` type. The `digitsof` and `precisionof` macros are defined in `decimal.h`.

The `digitsof` operator gives the number of significant digits of an object, and `precisionof` gives the number of decimal digits. That is,

```
digitsof(decimal(n,p)) = n
precisionof(decimal(n,p)) = p
```

The results of the `digitsof` and `precisionof` operators are integer constants.

Related Information

- “Fixed-Point Decimal Data Types” on page 46

Binary Expression

A *binary expression* contains two operands separated by one operator.

Not all binary operators have the same precedence. Table 8 on page 94 shows the order of precedence among operators. All binary operators have left-to-right associativity.

The order in which the operands of most binary operators are evaluated is not specified. Therefore, to ensure correct results, avoid creating binary expressions that depend on the order in which the compiler evaluates the operands.

Multiplication *

The `*` (multiplication) operator yields the product of its operands. The operands must have an arithmetic type. The result is not an lvalue. The usual arithmetic conversions are performed on the operands.

Related Information

- “Usual Arithmetic Conversions” on page 117

Division /

The / (division) operator yields the quotient of its operands. The operands must have an arithmetic type. The result is not an lvalue.

If both operands are positive integers and the operation produces a remainder, the remainder is ignored. Thus, the expression `7 / 4` yields the value `1` (rather than `1.75` or `2`).

For SAA-compliant compilers, the result of `-7 / 4` is `-1` with a remainder of `-3`.

The result is undefined if the second operand evaluates to `0`.

The result has the same type as the operands after the usual arithmetic conversions are performed.

Related Information

- "Usual Arithmetic Conversions" on page 117

Remainder %

The `%` (remainder) operator yields the remainder from the division of the left operand by the right operand. For example, the expression `5 % 3` yields `2`. The result is not an lvalue.

Both operands must have an integral type. If the right operand evaluates to `0`, the result is undefined. If either operand has a negative value, the result is such that the following expression always yields the value of `a`, if `b` is not `0` and `a/b` is representable:

```
( a / b ) * b + a % b;
```

The result has the same type as the operands after the usual arithmetic conversions are performed.

Related Information

- "Usual Arithmetic Conversions" on page 117

Addition +

The + (addition) operator yields the sum of its operands. Both operands must have an arithmetic type, or one operand must be a pointer to an object type, and the other operand must have an integral type.

When both operands have an arithmetic type, the usual arithmetic conversions are performed on the operands. The result has the type produced by the conversions on the operands and is not an lvalue.

When one of the operands is a pointer, the compiler converts the other operand to an address offset. The result is a pointer of the same type as the pointer operand. For example, after the addition, `ptr` will point to the third element of array:

```
int array[5];
int *ptr;
ptr = array + 2;
```

Related Information

- “Type Conversions” on page 118

Subtraction -

The - (subtraction) operator yields the difference of its operands. Both operands must have an arithmetic type, or the left operand must have a pointer type, and the right operand must have the same pointer type or an integral type.

When both operands have an arithmetic type, the usual arithmetic conversions are performed on the operands. The result has the type produced by the conversions on the operands and is not an lvalue.

When the left operand is a pointer and the right operand has an integral type, the compiler converts the value of the right to an address offset. The result is a pointer of the same type as the pointer operand.

If both operands are pointers to the same type, the compiler converts the result to an integral type that represents the number of objects separating the two addresses. Behavior is undefined if the pointers do not refer to objects in the same array.

Related Information

- “Type Conversions” on page 118

Bitwise Left and Right Shift << >>

The bitwise shift operators move the bit values of a binary object. The left operand specifies the value to be shifted. The right operand specifies the number of positions that the bits in the value are to be shifted. The result is not an lvalue.

The << (bitwise left shift) operator indicates that the bits are to be shifted to the left. The >> (bitwise right shift) operator indicates that the bits are to be shifted to the right.

Each operand must have an integral type. The compiler performs integral promotions on the operands. Then the right operand is converted to type `int`. The result has the same type as the left operand (after the arithmetic conversions).

If the right operand has a negative value or a value that is greater than or equal to the width in bits of the expression being shifted, the result is undefined.

If the right operand has the value `0`, the result is the value of the left operand (after the usual arithmetic conversions).

The << operator fills vacated bits with zeros. For example, if `l_op` has the value `4019`, the bit pattern (in 16-bit format) of `l_op` is:

```
0000111110110011
```

The expression `l_op << 3` yields:

```
0111110110011000
```

If the left operand has an `unsigned` type, the >> operator fills vacated bits with zeros. Otherwise, the compiler will fill the vacated bits of a signed value with a copy of the value's sign bit. For example, if `l_op` has the value `-25`, the bit pattern (in 16-bit format) of `l_op` is:

```
1111111111100111
```

The expression `l_op >> 3` yields:

```
1111111111111100
```

Relational `<` `>` `<=` `>=`

The relational operators compare two operands and determine the validity of a relationship. If the relationship stated by the operator is true, the value of the result is `1`. Otherwise, the value of the result is `0`.

The following table describes the relational operators.

Table 9. Relational Operators

Operator	Usage
`<`	Indicates whether the value of the left operand is less than the value of the right operand.
`>`	Indicates whether the value of the left operand is greater than the value of the right operand.
`<=`	Indicates whether the value of the left operand is less than or equal to the value of the right operand.
`>=`	Indicates whether the value of the left operand is greater than or equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type. The result has type `int`, and is not an lvalue.

If the operands have arithmetic types, the usual arithmetic conversions on the operands are performed.

When the operands are pointers, the result is determined by the locations of the objects to which the pointers refer. If the pointers do not refer to objects in the same array, the result is not defined.

Relational operators have left-to-right associativity. Therefore, the expression:

```
a < b <= c
```

is interpreted as:

```
(a < b) <= c
```

If the value of `a` is less than the value of `b`, the first relationship is true and yields the value `1`. The compiler then compares the value `1` with the value of `c`.

Equality == !=

The equality operators, like the relational operators, compare two operands for the validity of a relationship. The equality operators, however, have a lower precedence than the relational operators. If the relationship stated by an equality operator is true, the value of the result is `1`. Otherwise, the value of the result is `0`.

The following table describes the equality operators.

Table 10. Equality Operators

Operator	Usage
`==`	Indicates whether the value of the left operand is equal to the value of the right operand.
`!=`	Indicates whether the value of the left operand is not equal to the value of the right operand.

Both operands must have arithmetic types or be pointers to the same type, or one operand must have a pointer type and the other must be a pointer to void or `NULL`. The result has type `int`, and is not an lvalue.

If the operands have arithmetic types, the usual arithmetic conversions are performed on the operands.

If the operands are pointers, the result is determined by the locations of the objects to which the pointers refer.

If the operands are pointers to objects or incomplete types and they are both null pointers, they compare equal. If two pointers compare equal, they are either both null or they point to the same object.

You can also use the equality operators to compare pointers to members that are of the same type but do not belong to the same object.
The following expressions contain examples of equality and relational operators:

```
time < max_time == status < complete
letter != EOF
```

Bitwise AND &

The `&` (bitwise AND) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are `1`'s, the corresponding bit of the result is set to `1`. Otherwise, the corresponding result bit is set to `0`.

Both operands must have an integral type. The usual arithmetic conversions are performed on each operand. The result has the same type as the converted operands, and is not an lvalue.

Because the bitwise AND operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one bitwise AND operator.

The following example shows the values of `a`, `b`, and the result of `a & b` represented as 16-bit binary numbers:

bit pattern of `a`	`0000000001011100`
bit pattern of `b`	`0000000000101110`
bit pattern of `a & b`	`0000000000001100`

Bitwise Exclusive OR ^

The ^ (bitwise exclusive OR) operator compares each bit of its first operand to the corresponding bit of the second operand. If both bits are `1`'s or both bits are `0`'s, the corresponding bit of the result is set to `0`. Otherwise, the corresponding result bit is set to `1`.

Both operands must have an integral type. The usual arithmetic conversions on each operand are performed. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise exclusive OR operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one bitwise exclusive OR operator.

The following example shows the values of `a`, `b`, and the result of using the bitwise exclusive OR operator represented as 16-bit binary numbers:

bit pattern of `a`	`0000000001011100`
bit pattern of `b`	`0000000000101110`
bit pattern of `a ¬ b`	`0000000001110010`

Note: The bitwise exclusive OR may appear as a ¬ on your screen. On EBCDIC systems, and under C/VSE, the ^ symbol is represented by the ¬ symbol. For more information on these symbols, refer to the *LE/VSE C Run-Time Programming Guide*.

Bitwise Inclusive OR |

The | (bitwise inclusive OR) operator compares each bit of its first operand to the corresponding bit of the second operand and yields a value whose bit pattern shows which bits in either of the operands has the value `1`. If both of the bits are `0`, the result of that bit is `0`; otherwise, the result is `1`.

Both operands must have an integral type. The usual arithmetic conversions are performed on each operand. The result has the same type as the converted operands and is not an lvalue.

Because the bitwise inclusive OR operator has both associative and commutative properties, the compiler may rearrange the operands in an expression that contains more than one bitwise inclusive OR operator.

The following example shows the values of `a`, `b`, and the result of using the bitwise inclusive OR operator represented as 16-bit binary numbers:

bit pattern of `a`	`0000000001011100`
bit pattern of `b`	`0000000000101110`
bit pattern of `a	b`	`0000000001111110`

Logical AND &&

The `&&` (logical AND) operator indicates whether both operands have a nonzero value. If both operands have nonzero values, the result has the value `1`. Otherwise, the result has the value `0`.

Both operands must have a scalar type. The usual arithmetic conversions on each operand are performed. The result has type `int` and is not an lvalue.

The logical AND operator guarantees left-to-right evaluation of the operands. If the left operand evaluates to `0`, the right operand is not evaluated.

The following examples show how the expressions that contain the logical AND operator are evaluated:

Expression	Result
`1 && 0`	`0`
`1 && 4`	`1`
`0 && 0`	`0`

The following example uses the logical AND operator to avoid a divide-by-zero situation:

```
y && (x / y)
```

The expression `x / y` is not evaluated when `y` is `0`.

Note: The logical AND (`&&`) should not be confused with the address (&) operator or the bitwise AND (`&`) operator. For example:

`1 && 4` evaluates to 1
while
`1 & 4` evaluates to 0

Logical OR ||

The `||` (logical OR) operator indicates whether either operand has a nonzero value. If either operand has a nonzero value, the result has the value `1`. Otherwise, the result has the value `0`.

Both operands must have a scalar type. The usual arithmetic conversions are performed on each operand. The result has type `int` and is not an lvalue.

The logical OR operator guarantees left-to-right evaluation of the operands. If the left operand has a nonzero value, the right operand is not evaluated.

The following examples show how expressions that contain the logical OR operator are evaluated:

Expression	Result
`1 || 0`	`1`
`1 || 4`	`1`
`0 || 0`	`0`

The following example uses the logical OR operator to conditionally increment `y`:

```
++x || ++y;
```

The expression `++y` is not evaluated when the expression `++x` evaluates to a nonzero quantity.

Note: The logical OR (||) should not be confused with the bitwise OR (|) operator. For example:

`1 || 4` evaluates to 1
while
`1 | 4` evaluates to 5

Conditional Expression ? :

A *conditional expression* is a compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value `0` (the third expression).

The conditional expression contains one two-part operator. The `?` symbol follows the condition, and the `:` symbol appears between the two action expressions. All expressions that occur between the `?` and `:` are treated as one expression.

The first operand must have a scalar type. The second and third operands must have either arithmetic types, compatible structure types, compatible union types, or compatible pointer types. Types are compatible when they have the same type and are both qualified as `_Packed` or not packed, but not necessarily both qualified as `volatile` or `const`. Also, the second and third operands may be a pointer and a `NULL` pointer constant, or a pointer to an object (second operand) and a pointer to void (third operand).

The first operand is evaluated, and its value determines whether the second or third operand is evaluated:

- If the value is not equal to 0, the second operand is evaluated.
- If the value is equal to 0, the third operand is evaluated.

The result is the value of the second or third operand.

If the second and third operands are arithmetic types, the usual arithmetic conversions are performed on them.

For details on the types of the result, see Table 11 on page 114.

Table 11. Type of the Conditional Expression

Type of One Operand	Type of Other Operand	Type of Result
Arithmetic	Arithmetic	Arithmetic after usual arithmetic conversions
`struct/union` type	Compatible `struct/union` type	`struct/union` type with all the qualifiers on both operands
`void`	`void`	`void`
Pointer to type	Pointer to compatible type	Pointer to type with all the qualifiers specified for the type
Pointer to type	`NULL` pointer	Pointer to type
Pointer to object or incomplete type	Pointer to `void`	Pointer to `void` with all the qualifiers specified for the type

Conditional expressions have right-to-left associativity.

Examples

The following expression determines which variable has the greater value, `y` or `z`, and assigns the greater value to the variable `x`:

```
x = (y > z) ? y : z;
```

The preceding expression is equivalent to the following statement:

```
if (y > z)
   x = y;
else
   x = z;
```

The following expression calls the function `printf()`, which receives the value of the variable `c`, if `c` evaluates to a digit. Otherwise, `printf()` receives the character constant `'x'`.

```
printf(" c = %c\n", isdigit(c) ? c : 'x');
```

Assignment Expression

An *assignment expression* stores a value in the object designated by the left operand.

The left operand in all assignment expressions must be a modifiable lvalue. The type of the expression is the type of the left operand. The value of the expression is the value of the left operand after the assignment has completed. The result of an assignment expression is not an lvalue.

All assignment operators have the same precedence and have right-to-left associativity.

There are two types of assignment operators: simple assignment and compound assignment. The following sections describe these operators.

Simple Assignment =

The simple assignment operator stores the value of the right operand in the object designated by the left operand.

Both operands must have arithmetic types, the same structure type, or the same union type. Otherwise, both operands must be pointers to the same type, or the left operand must be a pointer and the right operand must be the constant `0` or `NULL`.

If both operands have arithmetic types, the system converts the type of the right operand to the type of the left operand before the assignment.

If the left operand is a pointer and the right operand is the constant `0`, the result is `NULL`.

Pointers to void can appear on either side of the simple assignment operator.

A packed structure or union can be assigned to a nonpacked structure or union of the same type, and a nonpacked structure or union can be assigned to a packed structure or union of the same type.

If one operand is packed and the other is not, the layout of the right operand is remapped to match the layout of the left. This remapping of structures may degrade performance. For efficiency, when you perform assignment operations with structures or unions, you should ensure that both operands are either packed or nonpacked.

Note: If you assign pointers to structures or unions, the objects they point to must both be either packed or nonpacked. See “Initializers of Pointers” on page 78 for more information on assignments with pointers.

You can assign values to operands with the type qualifier `volatile`. You cannot assign a pointer of an object with the type qualifier `const` to a pointer of an object without the type qualifier `const` such as in the following example:

```
const int *p1;
int *p2;
p2 = p1;  /* not allowed */

p1 = p2;  /* allowed */
```

The following example assigns the value of `number` to the member `employee` of the structure `payroll`:

```
payroll.employee = number;
```

The following example assigns in order the value `0` to `d`, the value of `d` to `c`, the value of `c` to `b`, and the value of `b` to `a`:

```
a = b = c = d = 0;
```

Note: The assignment (=) operator should not be confused with the equality comparison (==) operator. For example:

`if(x == 3)` evaluates to 1 if x is equal to three

while

`if(x = 3)` is taken to be true because `(x = 3)` evaluates to a non-zero value (3). The expression also assigns the value 3 to x.

Compound Assignment

The compound assignment operator consists of a binary operator and the simple assignment operator. It performs the operation of the binary operator on both operands and gives the result of that operation to the left operand.

If the left operand of the `+=` and `-=` operators is a pointer, the right operand must have an integral type; otherwise, both operands must have an arithmetic type.

Both operands of the `*=`, `/=`, and `%=` operators must have an arithmetic type.

Both operands of the `<<=`, `>>=`, `&=`, `¬=`, and `¦=` operators must have an integral type.

Note that the expression `a *= b + c` is equivalent to `a = a * (b + c)`, and **not** `a = a * b + c`.

The following table lists the compound assignment operators and shows an expression using each operator:

Operator	Example	Equivalent Expression
`+=`	`index += 2`	`index = index + 2`
`-=`	`*(pointer++) -= 1`	`*pointer = *(pointer++) - 1`
`*=`	`bonus *= increase`	`bonus = bonus * increase`
`/=`	`time /= hours`	`time = time / hours`
`%=`	`allowance %= 1000`	`allowance = allowance % 1000`
`<<=`	`result <<= num`	`result = result << num`
`>>=`	`form >>= 1`	`form = form >> 1`
`&=`	`mask &= 2`	`mask = mask & 2`
`¬=`	`test ¬= pre_test`	`test = test ¬ pre_test`
`|=`	`flag |= ON`	`flag = flag | ON`

Note: For more information on variant characters, refer to “Character Set” on page 7.

Comma Expression ,

A *comma expression* consists of two operands. Although the compiler evaluates both operands, the value and type of the right operand is the value and type of the expression. The left operand is evaluated, possibly producing side effects, and the value is discarded. The result of a comma expression is not an lvalue.

Both operands of a comma expression can have any type. All comma expressions have left-to-right associativity. The left operand is fully evaluated before the right operand.

If `omega` had the value `11`, the following example would increment `y` and assign the value `3` (the remainder when 11 is divided by 4) to `alpha`. The value of `y` is ignored.

```
alpha = (y++, omega % 4);
```

Any number of expressions separated by commas can form a single expression. The compiler evaluates the leftmost expression first. The value of the rightmost expression becomes the value of the entire expression. For example, the value of the following expression is `rotate(direction)`:

```
intensity++, shade * increment, rotate(direction);
```

Restrictions

You can place comma expressions within lists that contain commas (for example, argument lists and initializer lists). However, because the comma has a special meaning, you must place parentheses around comma expressions in these lists. The comma expression `t = 3, t + 2` is in the following function call:

```
f(a, (t = 3, t + 2), c);
```

The arguments to the function `f()` are the value of `a`, the value `5`, and the value of `c`.

Conversions

Differences in the types of operands cause *conversions*. A conversion changes the form of a value and its type; this can be implicit or explicit. For example, when you add values having different data types, the compiler converts the types of the objects to the same type before adding the values. Addition of a `short int` value and an `int` value causes the compiler to convert the `short int` value to the `int` type.

Conversions may occur, for example, when:

- A cast operation is performed
- An operand is prepared for an arithmetic or logical operation
- An assignment is made to an lvalue that has different type from the assigned value
- A prototyped function is given an argument that has a different type from the parameter
- The type of the expression specified on a function's `return` statement has a different type from the defined return type for the function

Usual Arithmetic Conversions

Type conversion is necessary to bring the operands of an expression to a common type or to extend short values to the integer size used in machine operations. The conversions depend on the specific operator and the type of the operand or operands. However, many operators perform similar conversions on operands of integer and floating-point types. These conversions are known as standard arithmetic conversions because they apply to the types of values ordinarily used in arithmetic.

A `char`, `short int`, `int bit field`, their signed or unsigned varieties, or an enumeration type, may be used in an expression wherever an `int` or `unsigned int` may be used. If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise, it is converted to an `unsigned int`. These are called *integral promotions*. All other arithmetic types are unchanged by integral promotions.

Conversions follow these arithmetic conversion rules:

- First, if either operand has type `long double`, the other operand is converted to type `long double`.
- Otherwise, if either operand has type `double`, the other operand is converted to type `double`.
- Otherwise, if either operand has type `float`, the other operand is converted to type `float`.
- Otherwise, if either operand has type `decimal`, the other operand is converted to type `decimal`.
- Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:
  - If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
  - Otherwise, if one operand has type `long int` and the other has type `unsigned int`, if a `long int` can represent all values of an `unsigned int`, the operand of type `unsigned int` is converted to `long int`; if a `long int` cannot represent all the values of an `unsigned int`, both operands are converted to `unsigned long int`.
  - Otherwise, if either operand has type `long int`, the other operand is converted to `long int`.
  - Otherwise, if either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
  - Otherwise, both operands have type `int`.

The remainder of this chapter discusses "Type Conversions," and outlines the path of each type of conversion.

Type Conversions

A type conversion occurs when:

- A value is explicitly cast to another type
- An operator converts the type of its operand or operands before performing an operation
- A value is passed as an argument to a function

The following sections outline the rules governing each kind of conversion.

In assignment operations, the type of the value being assigned is converted to the type of the variable receiving the assignment. C allows conversions by assignment between integer and floating-point types, even when the conversion entails loss of information.

In C/VSE, `int` types are equivalent to `long` types (they both occupy 4 bytes).

The methods of carrying out the conversions depend upon the type, as follows.

From Signed Integer Types: C converts a signed integer to a shorter signed integer by truncating the high-order bits, and converts a signed integer to a longer signed integer by sign-extension. Conversion of signed integers to floating-point values takes place without loss of information, except that some precision can be lost when a `long` value is converted to a `float`. To convert a signed integer to an unsigned integer, you must convert the signed integer to the size of the unsigned integer. The result is interpreted as an unsigned value.

Example

The following example shows the loss of precision when a signed integer is converted to a `float`.

EDCXRAAZ

```
 /* Example of loss of precision when converting
    from a signed integer to a float */

#include <stdio.h>

int main(void)
{

  int i;
  float f;
  double d;
  long double ld;

  i = 1234567890;
  f = i;
  d = i;
  ld = i;
  printf("i  = %d\n", i);
  printf("f  = %20.10Lf\n", f);
  printf("d  = %20.10Lf\n", d);
  printf("ld = %20.10Lf\n", ld);

}
```

This example produces the following output (note the loss of precision):

```
i  = 1234567890
f  = 1234567680.0000000470
d  = 1234567890.0000000470
ld = 1234567890.0000000000
```

When a signed integer is converted to a fixed-point decimal data type, the value becomes decimal(10,0) and the sign is unchanged.

Table 12 summarizes conversions from signed integer types.

Table 12 (Page 1 of 2). Summary of Conversions from Signed Integer Types

From	To	Method
`signed char`	`short`	Sign-extend
	`int`	Sign-extend
	`long`	Sign-extend
	`unsigned char`	Preserve pattern; high-order bit loses function as sign bit
	`unsigned short`	Sign-extend to `short`; convert `short` to `unsigned short`
	`unsigned long`	Sign-extend to `long`; convert `long` to `unsigned long`
	`float`	Sign-extend to `long`; convert `long` to `float`
	`double`	Sign-extend to `long`; convert `long` to `double`
	`long double`	Sign-extend to `long`; convert `long` to `long double`
	`fixed-point decimal`	Convert to `decimal(10,0)`; sign is unchanged

Table 12 (Page 2 of 2). Summary of Conversions from Signed Integer Types

From	To	Method
`short`	`signed char`	Preserve low-order byte
	`int`	Sign-extend
	`long`	Sign-extend
	`unsigned char`	Preserve low-order byte
	`unsigned short`	Preserve bit pattern; high-order bit loses function as sign bit
	`unsigned long`	Sign-extend to `long`; convert `long` to `unsigned long`
	`float`	Sign-extend to `long`; convert `long` to `float`
	`double`	Sign-extend to `long`; convert `long` to `double`
	`long double`	Sign-extend to `long`; convert `long` to `long double`
	`fixed-point decimal`	Convert to decimal(10,0); sign is unchanged
`int`	`signed char`	Preserve low-order byte
	`short`	Preserve low-order bytes
	`unsigned char`	Preserve low-order byte
	`unsigned short`	Preserve low-order bytes
	`unsigned long`	Preserve bit pattern; high-order bit loses function as sign bit
	`float`	Represent as a `float`; if the `long` cannot be represented exactly, some loss of precision occurs
	`double`	Represent as a `double`; if the `long` cannot be represented exactly, some loss of precision occurs
	`long double`	Represent as a `long double`; if the `long` cannot be represented exactly, some loss of precision occurs
	`fixed-point decimal`	Convert to decimal(10,0); sign is unchanged
`long`	`signed char`	Preserve low-order byte
	`short`	Preserve low-order bytes
	`unsigned char`	Preserve low-order byte
	`unsigned short`	Preserve low-order bytes
	`unsigned long`	Preserve bit pattern; high-order bit loses function as sign bit
	`float`	Represent as a `float`; if the `long` cannot be represented exactly, some loss of precision occurs
	`double`	Represent as a `double`; if the `long` cannot be represented exactly, some loss of precision occurs
	`long double`	Represent as a `long double`; if the `long` cannot be represented exactly, some loss of precision occurs
	`fixed-point decimal`	Convert to decimal(10,0); sign is unchanged

From Unsigned Integer Types: An unsigned integer is converted to a shorter unsigned or signed integer by truncating the high-order bits. An unsigned integer is converted to a longer unsigned or signed integer by setting the high-order bits to `0` (zero-extend). Conversion from `unsigned long` or `unsigned int` to `float` may result in a loss of some precision.

When an unsigned integer is converted to a signed integer of the same size, no change in the bit pattern occurs. However, the value changes if the sign bit is set.

An unsigned integer is converted to a positive fixed-point decimal value.

Table 13 summarizes conversions from unsigned integer types.

Table 13. Summary of Conversions from Unsigned Integer Types

From	To	Method
`unsigned char`	`signed char`	Preserve bit pattern; high-order bit becomes sign bit
	`short`	Zero-extend; preserve bit pattern
	`int`	Zero-extend; preserve bit pattern
	`long`	Zero-extend; preserve bit pattern
	`unsigned short`	Zero-extend; preserve bit pattern
	`unsigned int`	Zero-extend; preserve bit pattern
	`unsigned long`	Zero-extend; preserve bit pattern
	`float`	Convert to `long`; convert `long` to `float`
	`double`	Convert to `long`; convert `long` to `double`
	`long double`	Convert to `long`; convert `long` to `long double`
	`fixed-point decimal`	Convert to positive `decimal(10,0)`
`unsigned short`	`signed char`	Preserve low-order byte
	`short`	Preserve bit pattern; high-order bit becomes sign bit
	`int`	Zero-extend; preserve bit pattern
	`long`	Zero-extend; preserve bit pattern
	`unsigned char`	Preserve low-order byte
	`unsigned int`	Zero-extend
	`unsigned long`	Zero-extend
	`float`	Convert to `long`; convert `long` to `float`
	`double`	Convert to `long`; convert `long` to `double`
	`long double`	Convert to `long`; convert `long` to `long double`
	`fixed-point decimal`	Convert to positive `decimal(10,0)`
`unsigned int`	`signed char`	Preserve low-order byte
	`short`	Preserve low-order bytes
	`int`	Preserve bit pattern; high-order bit becomes sign
	`long`	Preserve bit pattern; high-order bit becomes sign
	`unsigned char`	Preserve low-order byte
	`unsigned short`	Preserve low-order bytes
	`float`	Convert `unsigned int` to `float`
	`double`	Convert `unsigned int` to `double`
	`long double`	Convert `unsigned int` to `long double`
	`fixed-point decimal`	Convert to positive `decimal(10,0)`
`unsigned long`	`signed char`	Preserve low-order byte
	`short`	Preserve low-order bytes
	`int`	Preserve bit pattern; high-order bit becomes sign
	`long`	Preserve bit pattern; high-order bit becomes sign
	`unsigned char`	Preserve low-order byte
	`unsigned short`	Preserve low-order bytes
	`float`	Convert `unsigned long` to `float`
	`double`	Convert `unsigned long` to `double`
	`long double`	Convert `unsigned long` to `long double`
	`fixed-point decimal`	Convert to positive `decimal(10,0)`

From Floating-Point Types: A `float` value converted to a `double` undergoes no change in value. A `double` converted to a `float` is represented exactly, if possible. If C cannot exactly represent the `double` value as a `float`, the number loses precision. If the value is too large to fit into a `float`, the number is undefined.

A floating-point value is converted to an integer value by converting to an `unsigned long`. The decimal fraction portion of the floating-point value is discarded in the conversion. If the result is still too large to fit, the result of the conversion is undefined.

When a floating-point value is converted to a fixed-point decimal data type and the integer part cannot be represented, an exception is raised. If the decimal portion cannot be represented, no exception is raised, but the decimal portion is truncated.

Table 14 summarizes conversions from floating-point types.

Table 14 (Page 1 of 2). Summary of Conversions from Floating-Point Types

From	To	Method
`float`	`signed char`	Convert to `long`; convert `long` to `signed char`
	`short`	Convert to `long`; convert `long` to `short`
	`int`	Truncate at decimal point; if result is too large to be represented as an `int`, result is undefined
	`long`	Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined
	`unsigned short`	Convert to `unsigned long`; convert `unsigned long` to `unsigned short`
	`unsigned int`	Truncate at decimal point; if result is too large to be represented as an `unsigned int`, result is undefined
	`unsigned long`	Truncate at decimal point; if result is too large to be represented as an `unsigned long`, result is undefined
	`double`	Represent as a `double`
	`fixed-point decimal`	Conversion is dependent on the size of the `decimal` type; the decimal may be truncated, or an exception may be raised if the result is too large
`double`	`signed char`	Convert to `float`; convert `float` to `char`
	`short`	Convert to `float`; convert `float` to `short`
	`int`	Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined
	`long`	Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined
	`unsigned short`	Convert to `unsigned long`; convert `unsigned long` to `unsigned short`
	`unsigned int`	Truncate at decimal point; if result is too large to be represented as an `unsigned int`, result is undefined
	`unsigned long`	Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined
	`float`	Represent as a float; if the `double` value cannot be represented exactly as a `float`, loss of precision occurs; if the value is too large to be represented in a `float`, the result is undefined
	`long double`	Represent as a `long double`
	`fixed-point decimal`	Conversion is dependent on the size of the `decimal` type; the decimal may be truncated, or an exception may be raised if the result is too large

Table 14 (Page 2 of 2). Summary of Conversions from Floating-Point Types

From	To	Method
`long double`	`signed char`	Convert to `double`; convert `double` to `float`; convert `float` to `char`
	`short`	Convert to `double`; convert `double` to `float`; convert `float` to `short`
	`int`	Truncate at decimal point; if result is too large to be represented as a `int`, result is undefined
	`long`	Truncate at decimal point; if result is too large to be represented as a `long`, result is undefined
	`unsigned short`	Convert to `double`; convert `double` to `unsigned long`; convert `unsigned long` to `unsigned short`
	`unsigned int`	Truncate at decimal point; if result is too large to be represented as an `unsigned int`, result is undefined
	`unsigned long`	Truncate at decimal point; if result is too large to be represented as an `unsigned long`, result is undefined
	`float`	Convert to `double`; represent as a `float`; if the `long double` value cannot be represented exactly as a `float`, loss of precision occurs; if the value is too large to be represented in a `float`, the result is undefined
	`double`	Represent as a `double`; If the result is too large to be represented as a `double`, result is undefined
	`fixed-point decimal`	Conversion is dependent on the size of the `decimal` type; the decimal may be truncated, or an exception may be raised if the result is too large

From Fixed-Point Decimal Data Types: A fixed-point decimal data type is converted to an integer value by discarding the fractional part and converting the integer part to the integer type.

When a fixed-point decimal data type is converted to a floating type, and the value being converted is outside the range of values that can be represented, the behavior is undefined. If the value being converted is within the range of values, but cannot be represented exactly, the result is truncated.

To and From Pointer Types: You can convert a pointer to one type of value to a pointer to a different type, however the result may not always be defined.

You can convert a pointer value to an integral value. The path of the conversion depends on the size of the pointer and the size of the integral type.

The conversion of an integer value to an address offset (in an expression with an integral type operand and a pointer type operand) is system dependent.

A pointer to a constant or a volatile object should never be assigned to a nonconstant or nonvolatile object.

A pointer to `void` can be converted to or from a pointer to any incomplete or object type.

You cannot convert a pointer to a floating type or a fixed-point decimal data type. Also, you can only convert a pointer to a function to a pointer to another function.

From Other Types: When you define a value using the `enum` type specifier, the value is treated as an `int`. Conversions to and from an `enum` value proceed as for the `int` type.

When a packed structure or union is assigned to a nonpacked structure or union of the same type, or an nonpacked structure is assigned to a packed structure or union of the same type, the layout of the right operand is remapped to match the layout of the left.

Table 15 summarizes conversions from other types.

Table 15. Summary of Conversions from Other Types

From	To	Method
`_packed struct`	`unpacked struct`	preserve bit pattern
`unpacked struct`	`_packed struct`	preserve bit pattern
`packed union`	`unpacked union`	preserve bit pattern
`unpacked union`	`packed union`	preserve bit pattern
`pointer to _packed struct`	`pointer to unpacked struct`	not valid
`pointer to unpacked struct`	`pointer to _packed struct`	not valid
`pointer to _packed union`	`pointer to unpacked union`	not valid
`pointer to unpacked union`	`pointer to _packed union`	not valid

The following example shows the memory content after a nonpacked structure is assigned to a packed structure. Refer to “Packed Structures” on page 55 for more information on the memory layout of packed structures.

EDCXRAA0

```
 /* Example shows memory content after assigning
    a nonpacked structure to a packed structure */

#include <stdio.h>

struct ss {
  char ch;
  int m;
  short sh;
  int n;};
struct ss st1 = { '\x12', 0x3456789A, 0xBCDE, 0xFEDCBA98 };
_Packed struct ss st2;
```

```
int main(void)
{
  int i, size;
  unsigned char *p;

  st2 = st1;      /* Assign a non-packed structure
                     to a packed structure              */

                  /* Print out the content of st1 in Hex */

  for (i = 0, size = sizeof(st1), p = (char *) &st1;
       i < size;
       ++i, ++p)
  printf("%.2X ", *p);
  putchar('\n');

                  /* Print out the content of st2 in Hex */

  for (i = 0, size = sizeof(st2), p = (char *) &st2;
       i < size;
       ++i, ++p)
  printf("%.2X ", *p);
  putchar('\n');
}
```

This example produces the following output:

```
12 00 00 00 34 56 78 9A BC DE 00 00 FE DC BA 98
12 34 56 78 9A BC DE FE DC BA 98
```

No other conversions between structure or union types are allowed.

The `void` type has no value, by definition. Therefore, it cannot be converted to any other type, nor can any value be converted to `void` by assignment. However, a value can be explicitly cast to `void`.

Chapter 5. C Language Statements

This chapter describes the following C language statements:

- Labels
- Block
- break
- continue
- do
- Expression
- for
- goto
- if
- Null
- return
- switch
- while

Labels

A *label* is an identifier that allows your program to transfer control to other statements within the same function. It identifies the statements to which control may be passed. A *label* is the only type of identifier that has function scope (see "Scope" on page 13). Control is transferred to the statement following the label by means of the `goto` or `switch` statements. A label has the form:

```
►►──identifier──:──statement──►◄
```

For example, the following are labels:

```
comment_complete: ;             /* Example of null statement label */
test_for_null: if (NULL == ptr) /* Example of statement label      */
```

The `case` and `default` labels, which have a specific use, are described later in this chapter.

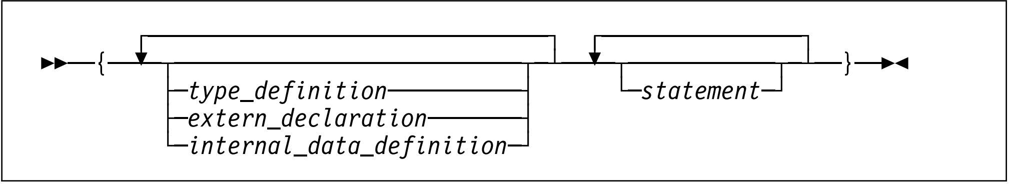
Related Information

- "goto" on page 136
- "switch" on page 140

Block

A *block statement* enables you to group any number of data definitions, declarations, and statements into one statement. When you enclose definitions, declarations, and statements within a single set of braces, everything within the braces is treated as a single statement. You can place a block wherever a statement is allowed.

The block statement has the form:



All definitions and declarations occur at the beginning of a block before the statements. Statements must follow the definitions and declarations. A block is considered a single statement.

If you redefine a data object inside a nested block, the inner object hides the outer object while the inner block is executed. Defining several variables that have the same identifier can make a program difficult to understand and maintain. Therefore, you should limit such redefinitions of identifiers within nested blocks.

If a data object is usable within a block, all nested blocks can use that data object (unless that data object identifier is redefined).

Initialization of an `auto` or `register` variable occurs each time the block is executed from the beginning. If you transfer control from one block to the middle of another block, initializations are not always performed. You cannot initialize an `extern` variable within a block.

Examples

The following example shows how the values of data objects change in nested blocks:

EDCXRAA1

```
 1    /* Example shows how values of data objects change
 2       in nested blocks */
 3
 4   #include <stdio.h>
 5
 6   int main(void)
 7   {
 8      int x = 1;                    /* Initialize x to 1  */
 9      int y = 3;
10
11      if (y > 0)
12      {
13         int x = 2;                 /* Initialize x to 2  */
14         printf("second x = %4d\n", x);
15      }
16      printf("first  x = %4d\n", x);
17   }
```

This example produces the following output:

```
second x =    2
first  x =    1
```

Two variables named `x` are defined in `main()`. The definition of `x` in line 8 retains storage throughout the execution of `main()`. However, because the definition of `x` in line 13 occurs within a nested block, line 14 recognizes `x` as the variable defined in line 13. Line 16 is not part of the nested block. Thus, line 16 recognizes `x` as the variable defined on line 8.

break

A *break statement* enables you to terminate and exit from a loop or `switch` statement from any point within the loop or `switch` other than the logical end. A `break` statement has the form:

```
►►──break──;──►◄
```

In a looping statement such as `do`, `for`, or `while` the `break` statement ends the loop and moves control to the next statement outside the loop. Within nested statements, the `break` statement ends only the smallest enclosing `do`, `for`, `switch`, or `while` statement.

In a `switch` body, the `break` statement ends the execution of the `switch` body and gives control to the next statement outside the `switch` body.

You can place a `break` statement only in the body of a looping statement (`do`, `for`, or `while`) or in the body of a `switch` statement.

Examples

The following example shows a `break` statement in the action part of a `for` statement. If the `i`th element of the array `string` is equal to `'\0'`, the `break` statement causes the `for` statement to end.

```
for (i = 0; i < 5; i++)
{
   if (string[i] == '\0')
      break;
   length++;
}
```

The following example shows a `break` statement in a nested looping statement. The outer loop sequences an array of pointers to strings. The inner loop examines each character of the string. When the `break` statement is executed, the inner loop ends and control returns to the outer loop.

EDCXRAA2

```
 /* Example of how to use break statement in a nested looping statement
    This example counts the characters in the strings that are
    part of an array of pointers to characters.  The count stops
    when one of the digits 0 through 9 is encountered
    and resumes at the beginning of the next string. */

#include <stdio.h>
#define  SIZE  3

int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;

   for (i = 0; i < SIZE; i++)        /* for each string    */
      for (pointer = strings[i]; *pointer != '\0'; ++pointer)
      {
         if (*pointer >= '0' && *pointer <= '9')
            break;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);
}
```

This example produces the following output:

```
letter count = 4
```

The following example is a `switch` statement that contains several `break` statements. Each `break` statement indicates the end of a specific clause and ends the execution of the `switch` statement.

EDCXRAA

```
 /* Example of a switch statement containing breaks */

#include <stdio.h>

enum {morning, afternoon, evening} timeofday = morning;
```

```
int main(void)
{
  switch (timeofday)
  {
     case (morning):
        printf("Good Morning\n");
        break;

     case (evening):
        printf("Good Evening\n");
        break;

     default:
        printf("Good Day, eh\n");
        break;
  }
}
```

Related Information

- "do" on page 133
- "for" on page 134
- "switch" on page 140
- "while" on page 144

continue

A *continue statement* enables you to terminate the current iteration of a loop. Program control is passed from the location in the body of the loop in which the `continue` statement is found to the end of the loop body. A `continue` statement has the form:

```
►►──continue──;──►◄
```

The `continue` statement ends the execution of the action part of a `do`, `for`, or `while` statement and moves control to the condition part of the statement. If the looping statement is a `for` statement, control moves to the third expression in the condition part of the statement, and then to the second expression (the test) in the condition part of the statement.

Within nested statements, the `continue` statement ends only the current iteration of the `do`, `for`, or `while` statement immediately enclosing it.

You can place a `continue` statement only within the body of a looping statement (`do`, `for`, or `while`).

Examples

The following example shows a `continue` statement in a `for` statement. The `continue` statement causes the system to skip over those elements of the array `rates` that have values less than or equal to `1`.

EDCXRAA3

```
 /* Example of a continue statement in a for statement */

#include <stdio.h>
#define  SIZE  5

int main(void)
{
   int i;
   static float rates[SIZE] = { 1.45, 0.05, 1.88, 2.00, 0.75 };

   printf("Rates over 1.00\n");
   for (i = 0; i < SIZE; i++)
   {
      if (rates[i] <= 1.00)  /*  skip rates <= 1.00  */
         continue;
      printf("rate = %.2f\n", rates[i]);
   }
}
```

This example produces the following output:

```
Rates over 1.00
rate = 1.45
rate = 1.88
rate = 2.00
```

The following example shows a `continue` statement in a nested loop. When the inner loop encounters a number in the array `strings`, that iteration of the loop is terminated. Execution continues with the third expression of the inner loop. (See "for" on page 134). The inner loop is terminated when the '`\0`' escape sequence is encountered.

EDCXRAA4

```
 /* Example of a continue statement in a nested loop
    This example counts the characters in strings that are part
    of an array of pointers to characters.  The count excludes
    the digits 0 through 9. */

#include <stdio.h>
#define  SIZE  3
```

```
int main(void)
{
   static char *strings[SIZE] = { "ab", "c5d", "e5" };
   int i;
   int letter_count = 0;
   char *pointer;
   for (i = 0; i < SIZE; i++)          /* for each string        */
      for (pointer = strings[i]; *pointer != '\0'; ++pointer)
      {
         if (*pointer >= '0' && *pointer <= '9')
            continue;
         letter_count++;
      }
   printf("letter count = %d\n", letter_count);
}
```

This example produces the following output:

```
letter count = 5
```

Compare the preceding program with the program “EDCXRAA” on page 130. The program on page 130 shows the use of the `break` statement but performs a similar function.

Related Information

- “do”
- “for” on page 134
- “while” on page 144

do

A *do statement* repeatedly executes a statement until the test expression evaluates to `0`. Because of the order of execution, the statement is executed at least once. It has the form:

```
►►──do──statement──while──(──expression──)──;──►◄
```

The body of the loop is executed before the `while` clause (the controlling expression) is evaluated. Further execution of the `do` statement depends on the value of the `while` clause. If the `while` clause does not evaluate to `0`, the statement is executed again. Otherwise, execution of the statement ends.

The controlling expression must be of scalar type.

A `break`, `return`, or `goto` statement can cause the execution of a `do` statement to end, even when the `while` clause does not evaluate to `0`.

Example

The following example prompts the user to enter a `1`. If the user enters a `1`, the statement ends execution. Otherwise, the statement displays another prompt.

EDCXRAA5

```
 /* Example of a do statement */

#include <stdio.h>

int main (void)
{
   int reply1;

   do
   {
      printf("Enter a 1\n");
      scanf("%d", &reply1);
   } while (reply1 != 1);
}
```

Related Information

- “break” on page 129
- “continue” on page 131
- “for”
- “while” on page 144

Expression

An *expression statement* contains an expression. Expressions are described in Chapter 4, “Expressions and Operators” on page 93. An expression statement has the form:

```
►►──┬──────────────┬──;──►◄
    └─expression───┘
```

An expression statement evaluates the given expression and is used to assign the value of the expression to a variable or to call a function.

Examples

```
printf("Account Number: \n");            /* A call to printf          */
marks = dollars * exch_rate;             /* An assignment to marks    */
(difference < 0) ? ++losses : ++gain;    /* A conditional increment   */
switches = flags ¦ BIT_MASK;             /* An assignment to switches */
```

Related Information

- Chapter 4, “Expressions and Operators” on page 93

for

A *for statement* enables you to:

- Repeatedly execute a statement
- Evaluate an expression prior to the first iteration of the statement (“initialization”)
- Specify an expression to determine whether or not the statement should be executed (“controlling part”)
- Evaluate an expression after each iteration of the statement

A `for` statement has the form:

```
►►──for──(──┬─────────────┬──;──┬─────────────┬──;──►
            └─expression1─┘     └─expression2─┘
►──┬─────────────┬──)──statement──►◄
   └─expression3─┘
```

expression1 is evaluated only once: before the statement is executed for the first time. You can use this expression to initialize a variable. If you do not want to evaluate an expression prior to the first iteration of the statement, you can omit this expression.

expression2 is evaluated before each execution of the statement. *expression2* must evaluate to a scalar type. If this expression evaluates to `0`, the `for` statement is terminated, and control moves to the statement following the `for` statement. Otherwise, the statement is executed. If you omit *expression2*, it will be as if the expression had been replaced by a nonzero constant, and the `for` statement will not be terminated by failure of this condition.

expression3 is evaluated after each execution of the statement. You can use this expression to increase, decrease, or reinitialize a variable. If you do not want to evaluate an expression after each iteration of the statement, you can omit this expression.

A `break`, `return`, or `goto` statement can cause the execution of a `for` statement to end, even when the second expression does not evaluate to `0`. If you omit *expression2*, you must use a `break`, `return`, or `goto` statement to stop the execution of the `for` statement.

Examples

The following `for` statement prints the value of `count` `20` times. The `for` statement initially sets the value of `count` to `1`. After each execution of the statement, `count` is incremented.

```
for (count = 1; count <= 20; count++)
   printf("count = %d\n", count);
```

For comparison purposes, the preceding example can be written using the following sequence of statements to accomplish the same task. Note the use of the `while` statement instead of the `for` statement.

```
count = 1;
while (count <= 20)
{
   printf("count = %d\n", count);
   count++;
}
```

The following `for` statement does not contain an initialization expression:

```
for (; index > 10; --index)
{
   list[index] = var1 + var2;
   printf("list[%d] = %d\n", index, list[index]);
}
```

The following `for` statement will continue executing until `scanf()` receives the letter `e`:

```
for (;;)
{
   scanf("%c", &letter);
   if (letter == '\n')
      continue;
   if (letter == 'e')
      break;
   printf("You entered the letter %c\n", letter);
}
```

The following `for` statement contains multiple initializations and increments. The comma operator makes this construction possible.

```
for (i = 0, j = 50; i < 10; ++i, j += 50)
{
    printf("i = %2d and j = %3d\n", i, j);
}
```

The following example shows a nested `for` statement. The outer statement is executed as long as the value of `row` is less than `5`. Each time the outer `for` statement is executed, the inner `for` statement sets the initial value of `column` to zero and the statement of the inner `for` statement is executed `3` times. The inner statement is executed as long as the value of `column` is less than `3`. This example prints the values of an array having the dimensions `[5][3]`:

```
for (row = 0; row < 5; row++)
   for (column = 0; column < 3; column++)
      printf("%d\n", table[row][column]);
```

Related Information

- "break" on page 129
- "continue" on page 131

goto

A *goto statement* causes your program to unconditionally transfer control to the statement associated with the label specified on the `goto` statement. A `goto` statement has the form:

```
►►──goto──identifier──;──►◄
```

The `goto` statement transfers control to the statement indicated by the identifier.

Notes

Use the `goto` statement sparingly. Because the `goto` statement can interfere with the normal top-to-bottom sequence of execution, it makes a program more difficult to read and maintain. Using a `goto` statement to jump into a loop may inhibit some optimization. Often, a `break` statement, a `continue` statement, or a function call can eliminate the need for a `goto` statement.

If you use a `goto` statement to transfer control to a statement inside of a loop or block, initializations of automatic storage for the loop do not take place and the result is undefined. The label must appear in the same function as the `goto`.

Examples

The following example shows a `goto` statement that is used to jump out of a nested loop. This function could be written without using a `goto` statement.

EDCXRAA6

```
 /* Example of a goto statement used to jump out of a nested loop */

#include <stdio.h>

void display(int matrix[3][3]);

int main(void)
{
   int matrix[3] [3]={1,2,3,4,5,2,8,9,10};
   display(matrix);
   return(0);
}

void display(int matrix[3][3])
{
   int i, j;

   for (i = 0; i < 3; i++)
      for (j = 0; j < 3; j++)
      {
         if ( (matrix[i][j] < 1) ¦¦ (matrix[i][j] > 6) )
            goto out_of_bounds;

         printf("matrix[%d][%d] = %d\n", i, j, matrix[i][j]);
      }
   return;
   out_of_bounds: printf("number must be 1 through 6\n");
}
```

This example produces the following output:

```
matrix[0][0] = 1
matrix[0][1] = 2
matrix[0][2] = 3
matrix[1][0] = 4
matrix[1][1] = 5
matrix[1][2] = 2
number must be 1 through 6
```

if

An *if statement* allows you to conditionally execute a statement when the specified test expression evaluates to a nonzero value. The expression must have a scalar type. Optionally, you can specify an `else` clause on the `if` statement. If the test expression evaluates to `0` and an `else` clause exists, the statement associated with the `else` clause is executed. An `if` statement has the form:

```
►►──if──(──expression──)──statement──┬──────────────────┬──►◄
                                     └─else──statement──┘
```

When `if` statements are nested and `else` clauses are present, a given `else` is associated with the closest preceding `if` statement within the same block.

Examples

The following example causes `grade` to receive the value `A` if the value of `score` is greater than or equal to `90`.

```
if (score >= 90)
   grade = 'A';
```

The following example displays `number is positive` if the value of `number` is greater than or equal to `0`. Otherwise, the example displays `number is negative`.

```
if (number >= 0)
   printf("number is positive\n");
else
   printf("number is negative\n");
```

The following example shows a nested `if` statement:

```
if (paygrade == 7)
   if (level >= 0 && level <= 8)
      salary *= 1.05;
   else
      salary *= 1.04;
else
   salary *= 1.06;
```

The following example shows an `if` statement that does not have an `else` clause. Because an `else` clause always associates with the closest `if` statement, braces may be necessary to force a particular `else` clause to associate with the correct `if` statement. In this example, omitting the braces would cause the `else` clause to associate with the nested `if` statement.

```
if (gallons > 0) {
   if (miles > gallons)
      mpg = miles/gallons;
}
else
   mpg = 0;
```

The following example shows an `if` statement nested within an `else` clause. This example tests multiple conditions. The tests are made in order of their appearance. If one test evaluates to a nonzero value, a statement executes and the entire `if` statement ends.

```
if (value > 0)
   ++increase;
else if (value == 0)
   ++break_even;
else
   ++decrease;
```

Null

The *null statement* performs no operation and has the form:

```
►►──;──►◄
```

You can use a null statement in a looping statement to show a nonexistent action or in a labeled statement to hold the label.

Example

The following example initializes the elements of the array `price`. Because the initializations occur within the `for` expressions, a statement is needed only to finish the `for` syntax; no operations are required.

```
for (i = 0; i < 3; price[i++] = 0)
   ;
```

return

A *return statement* terminates the execution of the current function and returns control to the caller of the function. A `return` statement has the form:

```
►►──return──┬──────────────┬──;──►◄
            └─expression─┘
```

A `return` statement is optional. If the system reaches the end of a function without encountering a `return` statement, control is passed to the caller as if a `return` statement without an expression were encountered.

A function can contain multiple `return` statements.

Value

If an expression is present on a `return` statement, the value of the expression is returned to the caller. If the data type of the expression is different from the data type of the function, conversion of the return value takes place as if the value of the expression were assigned to an object with the same data type as the function.

If an expression is not present on a `return` statement, the value of the `return` statement is not defined. If an expression is not given on a `return` statement and the calling function is expecting a value to be returned, the resulting behavior is undefined.

You cannot use a `return` statement with an expression when the function is declared as returning type `void`.

Examples

```
return;           /* Returns no value           */
return result;    /* Returns the value of result */
return 1;         /* Returns the value 1        */
return (x * x);   /* Returns the value of x * x  */
```

The following function searches through an array of integers to determine if a match exists for the variable `number`. If a match exists, the function `match()` returns the value of `i`. If a match does not exist, the function `match()` returns the value `-1` (negative one).

```
int match(int number, int array[ ], int n)
{
   int i;

   for (i = 0; i < n; i++)
      if (number == array[i])
         return (i);
   return(-1);
}
```

Related Information

- “Functions” on page 83
- “Expression” on page 134

switch

A *switch statement* enables you to transfer control to different statements within the `switch` body, depending on the value of the switch expression. The `switch` expression must have an integral type. Within the body of the `switch` statement, there are case labels that consist of a label, a case expression (that evaluates to an integral value), and statements, plus an optional default label. If the value of the `switch` expression equals the value of one of the case expressions, the statements following that case expression are executed. Otherwise, the default label statements, if any, are executed. A `switch` statement has the form:

```
►►──switch──(─expression─)──switch_body──►◄
```

For the various `case` expressions to execute different statements, the `switch_body` is enclosed in braces and can contain definitions, declarations, case clauses, and a `default` clause. Each `case` and `default` clause can contain statements.
The full form of a `switch_body` is:

```
      ┌───────────────────────────┐   ┌─────────────┐
►►──{─▼─┬─────────────────────────┬─┴───▼─┬─────────────┬─┴──►
        ├─type_definition─────────┤        └─case_clause─┘
        ├─extern_declaration──────┤
        └─internal_data_definition┘

                         ┌─────────────┐
►──┬────────────────┬────▼─┬─────────────┬─┴──}──►◄
   └─default_clause─┘      └─case_clause─┘
```

Note: The compiler does not initialize `auto` and `register` variables within the `type_definition`, `extern_declaration`, or `internal_data_definition`.

A `case_clause` contains a case label followed by one or more statements. A `case` label contains the word `case` followed by a constant expression and a colon. Anywhere you can place one `case` label, you can place multiple `case` labels. A `case_clause` has the form:

```
      ┌─────────────────────────────┐ ┌───────────┐
►►───▼─case──constant_expression──:─┴─▼─statement─┴──►◄
```

A `default_clause` contains a `default` label followed by one or more statements. You can place a `case` label on either side of the `default` label. The `default` label contains the word `default` followed by a colon. A `switch` statement can only have one `default` clause. A `default_clause` has the form:

```
                                                   ┌───────────┐
►►──┬──────────────┬──default──:──┬──────────────┬──▼─statement─┴──►◄
    └─case_label───┘              └─case_label───┘
```

The `switch` statement passes control to the statement following one of the labels or to the statement following the `switch` body. The value of the expression that precedes the `switch` body determines which statement receives control. This expression is called the *switch expression*.

The value of the `switch` expression is compared with the value of the expression in each `case` label. If a matching value is found, control is passed to the statement following the `case` label that contains the matching value. If the system does not find a matching value and a `default` label appears anywhere in the `switch` body, control passes to the `default` labeled statement. Otherwise, no part of the `switch` body is executed, and control is passed to the statement following the `switch` body.

If control passes to a statement in the `switch body`, control does not pass from the body until either a break statement is encountered or until the last statement in the switch body is executed.

Integral promotion is performed on the `switch` expression, and all `case` expressions are converted to the same type as the controlling expression.

Restrictions

The `switch` expression and the `case` expressions must have an integral type. The value of each `case` expression must be a unique constant integer expression.

A `case` expression must evaluate to an integral value, and the maximum number of case expressions is limited by `INT_MAX`.

Only one `default` label can occur in each `switch` statement.

Examples

The following `switch` statement contains several `case` clauses and one `default` clause. Each clause contains a function call and a `break` statement. The `break` statements prevent control from passing down through each statement in the `switch` body.

If the `switch` expression evaluated to `'/'`, the switch statement would call the function `divide()`. Control would then pass to the statement following the `switch` body.

```
char key;

printf("Enter an arithmetic operator\n");
scanf("%c",&key);

switch (key)
{
   case '+':
      add();
      break;

   case '-':
      subtract();
      break;

   case '*':
      multiply();
      break;

   case '/':
      divide();
      break;

   default:
      printf("The key you pressed is not valid\n");
      break;

}
```

If the switch expression matches a case expression, the statements following the case expression are executed until a `break` statement is encountered or until the end of the `switch` body is reached. In the following example, `break` statements are not present. If the value of `text[i]` is equal to `'A'`, all three counters are incremented. If the value of `text[i]` is equal to `'a'`, `lettera` and `total` are increased. Only `total` is increased if `text[i]` is not equal to `'A'` or `'a'`.

```
char text[] = "A switch statement test case";
int capa, lettera, total, size, i;

for (capa = 0, lettera = 0, total = 0, size = strlen(text), i = 0;
    i < size; i++)
{
switch(text[i])
  {
    case 'A':
      capa++;
    case 'a':
      lettera++;
    default:
      total++;
  }
}
```

The following switch statement performs the same statements for more than one case label:

EDCXRABI

```
 /* Example containing a switch statement that performs
    the same statement for more than one case label. */

#include <stdio.h>

int main(void)
{
  int month;

  /* Read in a month value */
  printf("Enter month: ");
  scanf("%d", &month);

  /* Tell what season it falls into */
  switch (month)
  {
     case 12:
     case 1:
     case 2:
        printf("month %d is a winter month\n", month);
        break;

     case 3:
     case 4:
     case 5:
        printf("month %d is a spring month\n", month);
        break;

     case 6:
     case 7:
     case 8:
        printf("month %d is a summer month\n", month);
        break;

     case 9:
     case 10:
     case 11:
        printf("month %d is a fall month\n", month);
        break;

     case 66:
     case 99:
     default:
        printf("month %d is not a valid month\n", month);
  }

  return(0);
}
```

If the expression month had the value 3, control would be passed to the statement:

```
printf("month %d is a spring month\n", month);
```

The break statement would pass control to the statement following the switch body.

Related Information

- “break” on page 129
- “Usual Arithmetic Conversions” on page 117

while

A *while statement* enables the compiler to repeatedly execute the body of a loop until the controlling expression evaluates to `0`. A `while` statement has the form:

```
►►──while──(─expression─)──statement──►◄
```

The expression is evaluated to determine whether or not the body of the loop should be executed. The expression must be a scalar type. If the expression evaluates to `0`, the statement terminates and the body of the loop is never executed. Otherwise, the body is executed. After the body has been executed, control is given once again to the expression. Further execution of the action depends on the value of the condition.

A `break`, `return`, or `goto` statement can cause the execution of a `while` statement to end, even when the condition does not evaluate to `0`.

Example

In the following program, `item[index]` triples each time the value of the expression `index` is less than `MAX_INDEX`. When `index` evaluates to `MAX_INDEX`, the `while` statement ends.

EDCXRAA7

```
/* Example of a while statement */

#define MAX_INDEX  (sizeof(item) / sizeof(item[0]))

#include <stdio.h>
int main(void)
{
   static int item[ ] = { 12, 55, 62, 85, 102 };
   int index = 0;

   while (index < MAX_INDEX)
   {
      item[index] *= 3;
      printf("item[%d] = %d\n", index, item[index]);
      ++index;
   }
}
```

This example produces the following output:

```
item[0] = 36
item[1] = 165
item[2] = 186
item[3] = 255
item[4] = 306
```

Related Information

- “break” on page 129
- “continue” on page 131

Chapter 6. Preprocessor Directives

This chapter describes the C preprocessor directives. *Preprocessing* is a step in the compilation process that:

- Replaces tokens in the current file with specified replacement tokens. A token is a series of characters delimited by a white space. The only white space allowed on a preprocessor directive is the space, horizontal tab, vertical tab, form feed, and comments. The newline character can also separate preprocessor tokens.
- Imbeds files within the current file.
- Conditionally compiles sections of the current file.
- Changes the line number of the next line of source and changes the file name of the current file.
- Generates diagnostic messages.

The preprocessor recognizes the following directives:

- `#define`
- `#undef`
- `#error`
- `#include`
- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#elif`
- `#endif`
- `#line`
- `#pragma`

Note: The # is not part of the name of the directive and can be separated from the name with a white space.

Preprocessor Directive Format

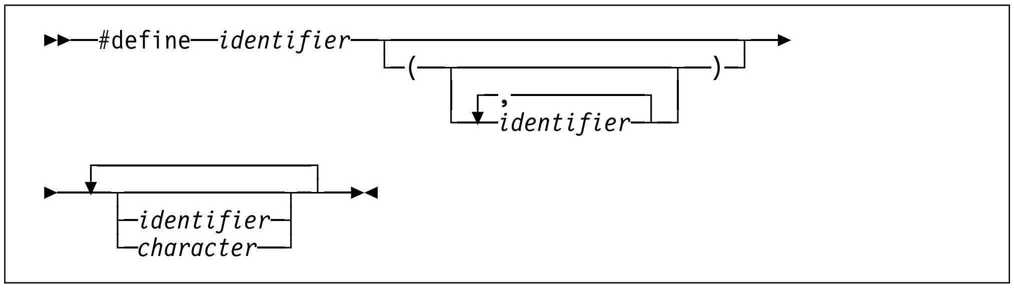
Preprocessor directives begin with the `#` token followed by a preprocessor keyword. The `#` token must appear as the first character that is not white space on a line.

A preprocessor directive ends at the newline character unless the last character of the line is the \ (backslash) character. If the \ character appears as the last character in the preprocessor line, the preprocessor interprets the \ and the newline character as a continuation marker, and interprets the following line as a continuation of the current preprocessor line.

Except for some `#pragma` directives, preprocessor directives can appear anywhere in a program.

#define

The *preprocessor define directive* defines macros and directs the preprocessor to replace all subsequent occurrences of a macro with specified replacement tokens. A preprocessor `#define` directive has the form:



The `#define` directive can contain an object-like macro definition or a function-like macro definition.

Object-Like Macro Definition

An *object-like macro definition* replaces a single identifier with the specified replacement tokens. They are called *feature test macros* when they are used to indicate the availability or existence of features. Feature test macros are used to control conditional compilation (see “Conditional Compilation” on page 157).

The following object-like definition causes the preprocessor to replace all subsequent instances of the identifier `COUNT` with the constant `1000`:

```
#define COUNT 1000
```

This definition causes the preprocessor to change the following statement (if the statement appears after the previous definition):

```
int arry[COUNT];
```

In the output of the preprocessor, the preceding statement appears as:

```
int arry[1000];
```

The following definition references the previously defined identifier `COUNT`:

```
#define MAX_COUNT COUNT + 100
```

The preprocessor replaces each subsequent occurrence of `MAX_COUNT` with `COUNT + 100`, which the preprocessor then replaces with `1000 + 100`.

Function-Like Macro Definition

Function-like macro definition: an identifier followed by a parenthesized parameter list and the replacement tokens. White space cannot separate the identifier (which is the name of the macro) and the left parenthesis of the parameter list. A comma must separate each parameter. For portability, do not have more than 31 parameters for a macro.

Function-like macro invocation: an identifier followed by a parenthesized list of arguments. A comma must separate each argument. Once the preprocessor identifies a function-like macro invocation, argument substitution takes place. A parameter in the replacement code is replaced by the corresponding argument. Any macro invocations contained in the argument itself are completely replaced before the argument replaces its corresponding parameter in the replacement code.

The following line defines the macro `SUM` as having two parameters `a` and `b` and the replacement tokens `(a + b)`:

```
#define SUM(a,b) (a + b)
```

This definition causes the preprocessor to change the following statements (if the statements appear after the previous definition):

```
c = SUM(x,y);
c = d * SUM(x,y);
```

In the output of the preprocessor, the preceding statements, appear as:

```
c = (x + y);
c = d * (x + y);
```

Notes:

1. A macro invocation must have the same number of arguments as the corresponding macro definition has parameters.
2. In the macro invocation argument list, commas that appear as character constants, in string constants or surrounded by parentheses, do not separate arguments.
3. The scope of a macro definition begins at the definition and does not end until a corresponding `#undef` directive is encountered. If there is no corresponding `#undef` directive, the scope of the macro definition lasts until the end of the compilation is reached.
4. A recursive macro is not fully expanded. For example, the definition

   ```
   #define x(a,b) x(a+1,b+1) + 4
   ```

   would expand

   ```
   x(20,10)
   ```

   to

   ```
   x(20+1,10+1) + 4
   ```

   rather than trying to expand the macro `x` over and over within itself. After the macro `x` is expanded, it is a function call to a function `x()`.
5. A definition is not required to specify replacement tokens. The following definition removes all instances of the token `static` from subsequent lines within the scope of this macro.

   ```
   #define static
   ```
6. You can change the definition of a defined identifier or macro with a second preprocessor `#define` directive only if the second preprocessor `#define` directive is preceded by a preprocessor `#undef` directive. See “#undef” on page 151. The `#undef` directive nullifies the first definition so that the same identifier can be used in a redefinition.

7. Within the text of the program, the preprocessor does not scan character constants or string constants for macro invocations.

Examples

The following program contains two macro definitions and a macro invocation that references both of the defined macros:

EDCXRAA8

```
 /* Example use of #define directives
    The example EDCXRAA9 shows the effect of preprocessor
    macro replacement on this program */

#define SQR(s)  ((s) * (s))
#define PRNT(a,b) \
{ printf("value 1 = %d\n", a); \
  printf("value 2 = %d\n", b) ; }

int main(void)
{
  int x = 2;
  int y = 3;

  PRNT(SQR(x),y);
}
```

After being interpreted by the preprocessor, the preceding program is replaced by code equivalent to the following:

EDCXRAA9

```
 /* Example use of #define directives
    This example shows the effect of preprocessor
    macro replacement on the program in example EDCXRAA8 */

int main(void)
{
  int x = 2;
  int y = 3;

  {
     printf("value 1 = %d\n", ( (x) * (x) ) );
     printf("value 2 = %d\n", y);
  }
}
```

This example produces the following output:

```
value 1 = 4
value 2 = 3
```

Related Information

- "#undef" on page 151
- "# Operator" on page 154
- "## Operator" on page 155

#undef

A *preprocessor undef directive* causes the preprocessor to end the scope of a preprocessor definition. A preprocessor #undef directive has the form:

```
►►──#undef──identifier──►◄
```

#undef is ignored if the identifier is not currently defined as a macro.

Examples

The following directives define BUFFER and SQR:

```
#define BUFFER 512
#define SQR(x) ((x) * (x))
```

The following directives nullify the preceding definitions:

```
#undef BUFFER
#undef SQR
```

Any occurrences of the identifiers BUFFER and SQR that follow these #undef directives are not replaced with any replacement tokens. Once the definition of a macro has been removed by an #undef directive, the identifier can be used in a new #define directive.

Related Information

- “#define” on page 148

Predefined Macros

The C language provides the following predefined macro names.

__LINE__	An integer representing the current source line number.
__FILE__	A character string literal containing the name of the source file.

In C/VSE, the __LINE__ macro represents the line number of the source file being processed, relative to the start of the source file. You can use the __FILE__ macro to determine which file is being processed. The following example illustrates the values of __LINE__ and __FILE__.

EDCXRABA

```
 /* Example use of __LINE__ and __FILE__
    File 1 of 2 - file 2 is EDCXRABX */

#include <stdio.h>
#include "edcxrabx.c"

int main(void)
{
   printf("At line %d of file %s\n", __LINE__, __FILE__);
   testsub();      /* Call to function defined in EDCXRABX */
⋮
}
```

EDCXRABX

```
 /* Example use of __LINE__ and __FILE__
    File 2 of 2 - file 1 is EDCXRABA */

testsub()
{
   printf("At line %d of file %s\n", __LINE__, __FILE__);
   return;
}
```

`__DATE__`
: A character string literal containing the date when the source file was compiled. The date will be in the form:

  `"Mmm dd yyyy"`

  where:

  `Mmm` represents the month in an abbreviated form (`Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, or `Dec`).

  `dd` represents the day. If the day is less than 10, the first `d` will be a blank character.

  `yyyy` represents the year.

`__TIME__`
: A character string literal containing the time when the source file was compiled. The time will be in the form:

  `"hh:mm:ss"`

  where:

  `hh` represents the hour.

  `mm` represents the minutes.

  `ss` represents the seconds.

`__TIMESTAMP__`
: Always returns the value:

  `"Mon Jan  1 01:01:01 1990"`

  (In some implementations of C, `__TIMESTAMP__` returns the date and time when the source file was last modified.)

`__STDC__`
: The integer `1`.

  Note: This macro is undefined if the `langlvl` pragma is set to anything other than `ANSI`.

Examples

The following `printf()` function calls will display the values of the predefined macros (`__LINE__`, `__FILE__`, `__TIME__`, and `__DATE__`) and will print a message indicating the program's conformance to ANSI standards based on `__STDC__`:

EDCXRABB

```
 /* Example use of predefined macros */

#pragma langlvl(ANSI)
#include <stdio.h>

#ifdef __STDC__
   #define CONFORM    "conforms"
#else
   #define CONFORM    "does not conform"
#endif

int main(void)
{
  printf("Line %d of file %s has been executed\n", __LINE__, __FILE__);
  printf("This file was compiled at %s on %s\n", __TIME__, __DATE__);
  printf("This program %s to ANSI standards\n", CONFORM);
}
```

Other Macros

`__LOCALE__`	A string literal representing the locale of the `LOCALE` compile-time option. If no `LOCALE` compile-time option was supplied, the macro is undefined.
	The following example illustrates how to set the run-time locale to the compile-time locale:
	`main()`
	`{`
	`   setlocale(LC_ALL, __LOCALE__);`
	`⋮`
	`}`

`__FILETAG__`	A string literal representing the character code set of the `filetag` pragma associated with the current file. If no `filetag` pragma is present, the macro is undefined.
`__CODESET__`	A string literal representing the character code set of the `LOCALE` compile-time option. If no `LOCALE` compile-time option was supplied, the macro is undefined.
`__COMPILER_VER__`	The compiler version. For C/VSE Version 1 Release 1 the value is X'11010000'.
`__TARGET_LIB__`	The target library version: X'11040000', for the LE/VSE Release 4 library.

Notes:

1. The predefined macro names consist of two underscore (_ _) characters immediately preceding the name, the name in uppercase letters, and two underscore characters immediately following the name.
2. The value of `__LINE__` will change during compilation as the compiler processes subsequent lines of your source program. Also, the values of `__FILE__` and

__FILETAG__ will change as the compiler processes any #include files that are part of your source program.

Restrictions

Predefined macro names cannot be the subject of a #define or #undef preprocessor directive.

Related Information

- "#define" on page 148
- "#undef" on page 151
- "#line" on page 161

# Operator

The # (single number sign) operator is used to convert a parameter of a function-like macro (see "Function-Like Macro Definition" on page 148) into a character string literal. If macro ABC is defined using the following directive:

```
#define ABC(x)   #x
```

all subsequent invocations of the macro ABC are expanded into a character string literal containing the argument passed to ABC. For example:

Invocation	Result of Macro Expansion
ABC(1)	"1"
ABC(Hello there)	"Hello there"

When you use the # operator in a function-like macro definition, the following rules apply:

1. A parameter in a function-like macro that is preceded by the # operator will be converted into a character string literal containing the argument passed to the macro.
2. White-space characters that appear before or after the argument passed to the macro will be deleted.
3. Multiple white-space characters imbedded within the argument passed to the macro will be replaced by a single space character.
4. If the argument passed to the macro contains a string literal and if a \ (backslash) character appears within the literal, a second \ character will be inserted before the original \ when the macro is expanded.
5. If the argument passed to the macro contains a " (double quotation mark) character, a \ character will be inserted before the " when the macro is expanded.
6. The conversion of an argument into a string literal occurs before macro expansion on that argument.
7. If more than one ## operator or # operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.
8. If the result of the replacement is not a valid character string literal, the behavior is undefined.

Examples

The following examples demonstrate the rules given in the preceding paragraph:

```
#define STR(x)       #x
#define XSTR(x)      STR(x)
#define ONE          1
```

Invocation	Result of Macro Expansion
`STR(\n "\n" '\n')`	`"\n \"\\n\" '\\n'"`
`STR(ONE)`	`"ONE"`
`XSTR(ONE)`	`"1"`
`XSTR("hello")`	`"\"hello\""`

Related Information

- "#define" on page 148
- "#undef" on page 151

## Operator

The `##` (double number sign) operator is used to concatenate two tokens in a macro invocation (text and/or arguments) given in a macro definition. If a macro `XY` was defined using the following directive:

```
#define XY(x,y)    x##y
```

the last token of the argument for `x` will be concatenated with the first token of the argument for `y`.

For example,

Invocation	Result of Macro Expansion
`XY(1, 2)`	`12`
`XY(Green, house)`	`Greenhouse`

When you use the `##` operator, the following rules apply:

1. The `##` operator cannot be the very first or very last item in the replacement list of a macro definition.
2. The last token of the item preceding the `##` operator is concatenated with the first token of the item following the `##` operator.
3. Concatenation takes place before any macros in arguments are expanded.
4. If the result of a concatenation is a valid macro name, it is available for further replacement even if it appears in a context in which it would not normally be available.
5. If more than one `##` operator and/or `#` operator appears in the replacement list of a macro definition, the order of evaluation of the operators is not defined.

Example

The following examples demonstrate the rules given in the preceding paragraph.

```
#define ArgArg(x, y)          x##y
#define ArgText(x)            x##TEXT
#define TextArg(x)            TEXT##x
#define TextText              TEXT##text
#define Jitter                1
#define bug                   2
#define Jitterbug             3
```

Invocation	Result of Macro Expansion
`ArgArg(var, 1)`	`"var1"`
`ArgText(var)`	`"varTEXT"`
`TextArg(var)`	`"TEXTvar"`
`TextText`	`"TEXTtext"`
`ArgArg(Jitter, bug)`	`3`

Related Information

- “#define” on page 148

#error

A *preprocessor error directive* causes the preprocessor to generate an error message and causes the compilation to fail. The `#error` directive has the form:

```
            ┌─────────────┐
►►──#error───▼─character──┴───►◄
```

You can use the `#error` directive as a safety check during compilation. For example, if your program uses *preprocessor conditional compilation directives* (see “Conditional Compilation” on page 157), you can place `#error` directives in the source file to make the compilation fail if a section of the program is reached that should be bypassed.

Example

The following directive generates the error message, `Error in TESTPGM1 - This section should not be compiled`:

```
#error Error in TESTPGM1 - This section should not be compiled
```

#include

A *preprocessor include directive* causes the preprocessor to replace the directive with the contents of the specified file. A preprocessor `#include` directive has the form:

```
►►──#include───┬─"file_name"─┬───►◄
               └─<file_name>─┘
```

If the file name is enclosed in double quotation marks, the preprocessor searches for it according to the search path for user include files. In the following example, `payroll.h` is a user include file.

```
#include "payroll.h"
```

If the file name is enclosed in the characters < and >, it is a system include file. The preprocessor searches for it according to the search path for system include files. In the following example, `stdio.h` is a system include file.

```
#include <stdio.h>
```

For information on include file searches, refer to the *C/VSE User's Guide*.

The preprocessor resolves macros on an `#include` directive. After macro replacement, the resulting token sequence must consist of a file name enclosed in either double quotation marks or the characters < and >. In the following example, `#define` is used to define a macro that represents the name of the C standard I/O header file. `#include` is then used to make the header file available to the C program.

```
#define  IO_HEADER   <stdio.h>
⋮
#include IO_HEADER   /* equivalent to specifying #include <stdio.h> */
⋮
```

If there are a number of declarations used by several files, you can place all these definitions in one file and `#include` that file in each file that uses the definitions. For example, the following file `defs.h` contains several definitions and an inclusion of an additional file of declarations:

```
/* defs.h */
#define TRUE 1
#define FALSE 0
#define BUFFERSIZE 512
#define MAX_ROW 66
#define MAX_COLUMN 80
int hour;
int min;
int sec;
#include "mydefs.h"
```

You can embed the definitions that appear in `defs.h` with the following directive:

```
#include "defs.h"
```

Conditional Compilation

A *preprocessor conditional compilation directive* causes the preprocessor to conditionally suppress the compilation of portions of source code. Such directives test a constant expression or an identifier to determine which tokens the preprocessor should pass on to the compiler and which tokens should be ignored. The directives are:

- `#if`
- `#ifdef`
- `#ifndef`
- `#else`
- `#elif`
- `#endif`

For each #if, #ifdef, and #ifndef directive, there are zero or more #elif directives, zero or one #else directive, and one matching #endif directive. All the matching directives are considered to be at the same nesting level.

You can have nested conditional compilation directives. If you have the following directives, the first #else will be matched with the #if directive.

```
#ifdef MACNAME
                 /*  tokens added if MACNAME is defined */
   #if TEST <=10
                 /* tokens added if MACNAME is defined and TEST <= 10 */
   #else
                 /* tokens added if MACNAME is defined and TEST >  10 */
   #endif
#else
                 /*  tokens added if MACNAME is not defined */
#endif
```

Each directive controls the block immediately following it. A block consists of all the tokens starting on the line following the directive and ending at the next conditional compilation directive at the same nesting level.

Each directive is processed in the order in which it is encountered. If an expression evaluates to zero, the block following the directive is ignored.

When a block following a preprocessor directive is to be ignored, the tokens are examined only to identify preprocessor directives within that block so that the conditional nesting level can be determined. All tokens other than the name of the directive are ignored.

Only the first block whose expression is nonzero is processed. The remaining blocks at that nesting level are ignored. If none of the blocks at that nesting level has been processed and there is an #else directive, the block following the #else directive is processed. If none of the blocks at that nesting level has been processed and there is no #else directive, the entire nesting level is ignored.

#if, #elif

The #if and #elif directives compare the value of the expression to zero. All macros are expanded, any defined() expressions are processed, and all remaining identifiers are replaced with the token 0.

The preprocessor #if directive has the following form:

```
►►──#if──constant_expression──►◄
```

The preprocessor #elif directive has the following form:

```
►►──#elif──constant_expression──►◄
```

The expressions that are tested must be integer constant expressions that follow these rules:

- No casts are performed.
- The constant expression can contain the unary operator `defined`. This can be used only with the preprocessor keyword `#if`. The following expressions evaluate to `1` if the *identifier* is defined in the preprocessor; otherwise, to `0`:

  ```
  defined identifier
  defined(identifier)
  ```

- The expression can contain defined macros.
- Any arithmetic is performed using `long ints`.

If the constant expression evaluates to a nonzero value, the tokens that immediately follow the condition are passed on to the compiler.

#ifdef

The `#ifdef` directive checks for the existence of macro definitions.

If the identifier specified is defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The preprocessor `#ifdef` directive has the following form:

```
►►──#ifdef──identifier──►◄
```

The following example defines `MAX_LEN` to be `75` if `EXTENDED` is defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be `50`.

```
#ifdef EXTENDED
   #define MAX_LEN 75
#else
   #define MAX_LEN 50
#endif
```

#ifndef

The `#ifndef` directive checks for the existence of macro definitions.

If the identifier specified is not defined as a macro, the tokens that immediately follow the condition are passed on to the compiler.

The preprocessor `#ifndef` directive has the following form:

```
►►──#ifndef──identifier──►◄
```

The following example defines `MAX_LEN` to be `50` if `EXTENDED` is not defined for the preprocessor. Otherwise, `MAX_LEN` is defined to be `75`.

```
#ifndef EXTENDED
   #define MAX_LEN 50
#else
   #define MAX_LEN 75
#endif
```

#else

If the condition specified in the #if, #ifdef, or #ifndef directive evaluates to 0, and the conditional compilation directive contains a preprocessor #else directive, the source text located between the preprocessor #else directive and the preprocessor #endif directive is selected by the preprocessor to be passed on to the compiler.

The preprocessor #else directive has the form:

```
►►──#else──►◄
```

#endif

The preprocessor #endif directive ends the conditional compilation directive. It has the form:

```
►►──#endif──►◄
```

The following example shows how you can nest preprocessor conditional compilation directives.

```
#if defined(TARGET1)
   #define SIZEOF_INT 16
   #ifdef PHASE2
      #define MAX_PHASE 2
   #else
      #define MAX_PHASE 8
   #endif
#elif defined(TARGET2)
   #define SIZEOF_INT 32
   #define MAX_PHASE 16
#else
   #define SIZEOF_INT 32
   #define MAX_PHASE 32
#endif
```

The following program contains preprocessor conditional compilation directives.

EDCXRABC

```
 /* Example containing conditional compilation directives */

int main(void)
{
   static int array[ ] = { 1, 2, 3, 4, 5 };
   int i;
   for (i = 0; i <= 4; i++)
   {
      array[i] *= 2;
#if TEST >= 1
   printf("i = %d\n", i);
   printf("array[i] = %d\n", array[i]);
#endif
   }
}
```

#line

A *preprocessor line control directive* causes the compiler to view the line number of the next source line as the specified number. A preprocessor `#line` directive has the form:

```
►►──#line──┬─decimal_constant──┬─────────────┬─┬──►◄
           │                   └─"file_name"─┘ │
           └─characters────────────────────────┘
```

A file name specification enclosed in double quotation marks can follow the line number. If you specify a file name, the compiler views the next line as part of the specified file. If you do not specify a file name, the compiler views the next line as part of the current source file.

The token sequence on a `#line` directive is subject to macro replacement. After macro replacement, the resulting character sequence must consist of a decimal constant, optionally followed by a file name enclosed in double quotation marks.

Note: The `#line` directives are ignored by the compiler when either the `EVENTS` compile-time option or the `TEST` compile-time option is in effect. Do not use any `#line` directives before a `#pragma options` directive that contains the `TEST` option.

Examples

You can use `#line` control directives to make the compiler provide more meaningful error messages. The following program uses `#line` control directives to give each function an easily recognizable line number:

EDCXRABD

```
 /* Example containing preprocessor line control directives */

#include <stdio.h>
#define LINE200 200

int main(void)
{
   func_1();
   func_2();
}

#line 100
func_1()
{
   printf("Func_1 - the current line number is %d\n",__LINE__);
}

#line LINE200
func_2()
{
   printf("Func_2 - the current line number is %d\n",__LINE__);
}
```

This example produces the following output:

```
Func_1 - the current line number is 102
Func_2 - the current line number is 202
```

# (Null Directive)

The *null directive* performs no action. It consists of a single # on a line of its own.

Example

In the following example, if MINVAL is a defined macro name, no action is performed. If MINVAL is not a defined identifier, it is defined as the value 1.

```
#ifdef MINVAL
  #
#else
  #define MINVAL 1
#endif
```

#pragma

Along with the pragmas defined under SAA C, C/VSE also supports additional pragmas. The following are the SAA pragmas and additional C/VSE pragmas.

SAA Standard pragmas

chars
comment
langlvl
linkage
map
page
pagesize
skip
strings
subtitle
title

Additional C/VSE pragmas

checkout
csect
filetag
inline
longname
margins
options
runopts
sequence
target
variable

A pragma is an implementation-defined instruction to the compiler. It has the general form:

```
              ┌──────────────────┐
►►──#pragma───▼─character-sequence─┴───►◄
```

character-sequence is a series of characters providing a specific compiler instruction and arguments, if any.

The `character-sequence` on a pragma is not subject to macro substitutions. You can specify more than one pragma option for a single `#pragma` instruction, for example:

```
#pragma chars(signed) comment(compiler)
```

Table 16 lists the restrictions when you are using `#pragma` directives. A blank entry in the table indicates no restrictions.

Table 16 (Page 1 of 2). Restrictions on `#pragma`*s*

#pragma	Restriction on Number of Occurrences	Restriction on Placement
chars	Once.	On the first `#pragma` directive, and before any code or directive, except for the `#pragmas filetag`, `longname`, `langlvl` or `target`, which may precede this directive.

Table 16 (Page 2 of 2). Restrictions on #pragmas

#pragma	Restriction on Number of Occurrences	Restriction on Placement
checkout		
comment	The copyright directive can appear only once.	The copyright directive must appear before any C code.
csect	Twice. Once for code and once for static data.	
filetag	Once per file scope.	On the first #pragma directive, and before any code or directive, except for all conditional compilation directives (such as #if or #ifdef) which may precede this directive.
inline		At file scope.
langlvl	Once.	On the first #pragma directive, and before any code or directive, except for the #pragmas filetag, longname, chars or target, which may precede this directive.
linkage	Can appear more than once for each function, as long as one #pragma does not contradict another #pragma.	
longname	Once.	On the first #pragma directive, except for #pragmas filetag, chars, langlvl or target, which may precede this directive.
map		
margins		
options		Before any C code
page		
pagesize		
runopts		Before any C code.
sequence		
skip		
strings	Once.	Before any C code.
subtitle		
target	Once.	On the first #pragma directive, and before any code or directive, except for #pragmas filetag, chars, langlvl, or longname, which may precede this directive.
title		
variable		

Examples of #pragma Directives

```
#pragma langlvl(SAA)
#pragma title("SAA pragma example")
#pragma pagesize(55)
#pragma map(ABC, "A$$BC@")
```

chars

```
►►──#pragma──chars──(──┬─signed───┬──)──►◄
                       └─unsigned─┘
```

The `#pragma chars` directive specifies that the compiler is to treat all `char` types as `signed` or `unsigned`.

For C/VSE, the default for `char` types is `unsigned`.

checkout

The C/VSE directive `#pragma checkout` is an addition to the SAA Standard.

```
►►──#pragma──checkout──(──┬─resume──┬──)──►◄
                          └─suspend─┘
```

With `#pragma checkout`, you can suspend the diagnostics that the `CHECKOUT` compile-time option performs during specific portions of your program. You can then resume the same level of diagnostics later in the file.

Nested `#pragma checkout` directives are allowed and behave as shown in the following example:

```
/*  Assume CHECKOUT(PPTRACE) had been specified  */
#pragma checkout(suspend)  /*  No CHECKOUT diagnostics are performed */
   ...
#pragma checkout(suspend)  /*  No effect  */     ─┐
   ...                                            │
#pragma checkout(resume)   /*  No effect  */     ─┘
   ...
#pragma checkout(resume)   /*  CHECKOUT(PPTRACE) diagnostics continue  */
```

comment

```
►►──#pragma──comment──(──┬─compiler────────────────────────────┬──►
                         ├─date────────────────────────────────┤
                         ├─timestamp───────────────────────────┤
                         └─┬─copyright─┬──┬──────────────────┬─┘
                           └─user──────┘  └─,─"characters"─┘

►──)──►◄
```

The `#pragma comment` directive places a comment into the object file.

The `comment` type may be:

`compiler`	Places the name and version of the compiler into the end of the created object module.
`date`	Places the date and time of compilation into the end of the created object module.
`timestamp`	Places the text `Mon Jan  1 01:01:01 1990` into the end of the created object module. (In some implementations of C, `timestamp` places the last modification date and time of the source into the end of the created object module.)
`copyright`	Places the text specified by the character field into the produced object module. This text is loaded into memory when the program is run.
`user`	Places the text specified by the character field into the produced object module. This text is not loaded into memory when the program is run. It is placed at the end of the object file, and on `END` records, occupying columns 34 to 71.

C/VSE Addition to SAA Standard

The characters in the character field must be enclosed in double quotation marks. The maximum number of characters allowed is equal to the maximum number of characters allowed in C/VSE string literals, 4K.

Note: When the linkage editor is creating the executable phase, the `END` records are stripped from the module. If the number of characters exceeds the space available on one record, you can create additional `END` records by placing a `NULL` in column 71. This increases the space for comments.

csect

The C/VSE directive `#pragma csect` is additional to the SAA Standard.

```
►►──#pragma──csect──(──┬─CODE───┬──,──"name"──)──►◄
                       └─STATIC─┘
```

The `#pragma csect` directive identifies the name for either the code or static control section (CSECT).

`code`	specifies the CSECT containing the executable code (C functions) and constant data.
`static`	designates the CSECT containing all program variables with the `static` storage class and all character strings.

The *name* is enclosed in double quotation marks; it is the name used for the applicable (code or static) CSECT. It should not exceed 8 characters, and is not mapped in any way, including uppercasing. The name must not conflict with the name of an exposed name (external function or object) in a source file or with the name in another `#pragma csect` directive or `#pragma map` directive. For example, the name for the code CSECT must differ from the name for the static CSECT.

At most, two `#pragma csect` directives can appear in a source program: one for the code CSECT and one for the static CSECT. If a `#pragma csect` directive is not supplied, the CSECT is unnamed (this is also known as private code (PC)). The compiler cannot automatically create a name that does not conflict with a user name. Use the `CSECT` compile-time option to check that you have included the appropriate `#pragma csect` directives in your program.

filetag

The C/VSE directive `#pragma filetag` is additional to the SAA Standard.

```
►►──#pragma──filetag──(──"code_set_name"──)──►◄
```

The `#pragma filetag` directive is used to specify the code set in which the source code was entered. Since the `#` character is variant between code sets, it is recommended that you use the trigraph representation `??=` instead of `#` as illustrated below.

The `#pragma filetag` directive must appear at most once per source file, and must appear before the first statement or directive, except for all conditional compilation directives, which may precede this directive. For example:

```
??=ifdef COMPILER_VER            /* This is allowed            */
  ??=pragma filetag ("code set")  /* include pragma filetag only */
??=endif                        /* when you compile with the    */
                                /* C/VSE compiler              */
```

If there are comments before the pragma, they will not be translated to the code page associated with the `LOCALE` option.

See the *LE/VSE C Run-Time Programming Guide* for details on using this directive with the `LOCALE` option.

inline

The C/VSE directive `#pragma inline` is additional to the SAA Standard.

```
►►──#pragma──┬─inline───┬──(──function──)──►◄
             └─noinline─┘
```

The `#pragma inline` directive specifies whether or not the *function* is to be inlined. The pragma can be anywhere in the source but must be at file scope. `#pragma inline` has no effect if the `inline` compile-time option is not specified.

If you specify `#pragma inline`, the function is inlined on every call. The function is inlined in both selective (`NOAUTO`) and automatic (`AUTO`) mode.

If you specify `#pragma noinline`, the function is never inlined when it is called. This pragma has no effect when `NOAUTO` is specified.

Example

EDCXRABE

```
 /* Example of #pragma inline */

#pragma csect(code,"MYCFILE")
#pragma csect(static,"MYSFILE")
#pragma options(INLINE)

#include <stdio.h>

static int (writerecord) (int, char *);

#pragma inline (writerecord)

int main()
{
   int chardigit;
   int digit;

   printf("Enter a digit\n");
   chardigit = getchar();

   digit = chardigit - '0';

   if (digit < 0 || digit > 9)
     {
      printf("invalid digit\n");
      exit(99);
     }

   switch(digit)
     {
      case 0:
         writerecord(0, "entered 0");
         break;
      case 1:
         writerecord(1, "entered 1");
         break;
      default:
         writerecord(9, "entered other");
     }
}
```

```
static int writerecord (int digit, char *phrase)
{
   switch (digit)
     {
      case 0:
         printf("writerecord 0: ");
         printf("%s\n", phrase);
         break;
      case 1:
         printf("writerecord 1: ");
         printf("%s\n", phrase);
         break;
      case 2:
         printf("writerecord 2: ");
         printf("%s\n", phrase);
         break;
      case 3:
         printf("writerecord 3: ");
         printf("%s\n", phrase);
         break;
      default:
         printf("writerecord X: ");
         printf("%s\n", phrase);
      }
    return 0;
}
```

For more information on how to use `inline` and `noinline`, refer to the *LE/VSE C Run-Time Library Reference*.

langlvl

```
►►──#pragma──langlvl──(──┬─ANSI─────┬──)──►◄
                         ├─SAA──────┤
                         ├─SAAL2────┤
                         └─EXTENDED─┘
```

The `#pragma langlvl` directive specifies that use of language elements or library functions that are not of the specified level (`ANSI`, `SAA`, `SAAL2`, or `EXTENDED`) are to be flagged by the compiler. The compiler defines preprocessor variables that are used in header files to define the language level. This pragma must be before any statements in a file.

The language levels are as follows:

`ANSI`	Defines the preprocessor variables `__ANSI__` and `__STDC__`, and undefines other `langlvl` variables.
`SAA`	Defines the preprocessor variables `__SAA__` and `__SAA_L2__`, and undefines other `langlvl` variables.
`SAAL2`	Defines the preprocessor variable `__SAA_L2__`, and undefines other `langlvl` variables.
`EXTENDED`	Defines the preprocessor variable `__EXTENDED__`, and undefines other `langlvl` variables.

For C/VSE, the default language level is `EXTENDED`.

linkage

```
►►──#pragma──linkage──(identifier,──┬─OS────────┬──)──►◄
                                     ├─FETCHABLE─┤
                                     ├─PLI───────┤
                                     └─COBOL─────┘
```

The `#pragma linkage` directive identifies the entry point of modules used in interlanguage calls. The *identifier* either identifies the name of the function that is to be the entry point of the module, or identifies a `typedef` that will be used to define the entry point.

C/VSE Addition to SAA Standard

The `#pragma linkage` directive also designates other entry points within a program that can be used in a `fetch` operation.

A `typedef` can be used in a `#pragma linkage` directive to associate a specific linkage convention with the `typedef` of a function.

```
typedef void func_t(void);
#pragma linkage (func_t,OS)
```

In the example, the `#pragma linkage` directive associates the `OS` linkage convention with the `typedef func_t`.

This `typedef` can be used in C declarations wherever a function type specifies the type function of `OS` linkage type. The following are the linkage entry points:

`FETCHABLE`	Specifies a name, other than `main`, as an entry point within the program. This pragma also indicates that this name (*identifier* in the syntax diagram) can be used in a `fetch` operation. See the *LE/VSE C Run-Time Library Reference* for more details on the use of the `fetch` library function.
`OS`	Designates an entry point (*identifier* in the syntax diagram) as an `OS` linkage entry point. The `OS` linkage is the basic linkage convention used by the operating system.
`PLI`	Designates an entry point (*identifier* in the syntax diagram) as a PL/I linkage entry point. You can only call modules that have been written in LE/VSE-conforming languages. See the *LE/VSE Writing Interlanguage Communication Applications* for information on interlanguage calls.
`COBOL`	Designates an entry point (*identifier* in the syntax diagram) as a COBOL linkage entry point. See the *LE/VSE Writing Interlanguage Communication Applications* for more details on C modules calling COBOL modules.

longname

The C/VSE directive `#pragma longname` is additional to the SAA Standard.

```
►►──#pragma──┬─longname───┬──►◄
             └─nolongname─┘
```

The `#pragma longname` directive specifies that the compiler is to generate long and mixed case names in the object module produced by the compiler. These names can be up to 255 characters in length. If you use the `longname` directive, you must use the prelinker. If you specify the `NOLONGNAME` compile-time option, the `longname` directive is ignored.

If you have more than one preprocessor directive, `#pragma longname` may be preceded only by `#pragma filetag`, `#pragma chars`, `#pragma langlvl`, and `#pragma target`. Some directives, such as `#pragma variable` and `#pragma linkage` are sensitive to the name handling.

The `nolongname` pragma directive specifies that the compiler is to generate truncated and uppercase names in the object module produced by the compiler. More details on external name mapping are in "External Name Mapping in C/VSE" on page 18. If you specify the `LONGNAME` compile-time option, the `nolongname` directive is not used. If you have more than one preprocessor directive, `#pragma nolongname` must be the first one.

map

```
►►──#pragma──map──(──identifier──,──"name"──)──►◄
```

The `#pragma map` directive associates an external name (*name*) with a C name (*identifier*). If you use the `#pragma map` directive, the C name in the source file is not visible in the object deck, and the map name represents the object in the object deck.

C/VSE Addition to SAA Standard

name should be enclosed in double quotation marks. Its length must not exceed 8 characters, because external names in object modules can be 8 characters at most without the `LONGNAME` compile-time option. The name is kept as specified on the `#pragma map` directive in mixed case. The name must not conflict with the name in another `#pragma csect` directive.

The map name is an external name and must not be used in the source file to reference the object. If you use the map name in the source file to access the corresponding C object, the compiler treats it as a new identifier.

The compiler produces an error message if more than one map name is given to an identifier. Two different C identifiers can have the same map name.

margins

The C/VSE directive `#pragma margins` is additional to the SAA Standard.

```
►►──#pragma──┬─margins──(──m──,──┬─n─┬──)─┬──►◄
             │                   └─*─┘    │
             └─nomargins──────────────────┘
```

The `#pragma margins` directive specifies the margins in the source file that are to be scanned for input to the compiler. The `#pragma nomargins` directive specifies that the entire input source record is to be scanned for input to the compiler.

The margin setting specified by the `#pragma margins` directive applies only to the source file or include file in which it is found and has no effect on other `#include` files. The `#pragma margins` and the `#pragma nomargins` directives come into effect on the line following the directive, and remain in effect until another `#pragma margins` or `#pragma nomargins` directive is encountered or until the end of the file is reached.

If you use the compile-time `MARGINS` or `NOMARGINS` option with the `#pragma margins` or `#pragma nomargins` directives, the `#pragma` directives override the compile-time options. The compile-time option specified will be in effect up to, and including, the `#pragma margins` or `#pragma nomargins` directive.

The default setting is `MARGINS(1,72)` for fixed-length records, and `NOMARGINS` for variable-length records.

In the syntax diagram, the following parameters are specified:

m The first column of the source input containing a valid C program. The value of *m* must be:

    Greater than `0` and less than `32768`
and
    Less than or equal to the value of *n*

n The last column of the source input containing a valid C program. The value of *n* must be greater than `0` and less than `32768`

An asterisk (*) can be assigned to *n* indicating the last column of the input record. For example, if you specify `#pragma margins(8,*)`, the compiler scans from column `8` to the end of the record for input source statements.

You can use `#pragma margins` and `#pragma sequence` together. If they reserve the same columns, `#pragma sequence` has priority and the columns are reserved for sequence numbers. For example, if the columns reserved for margin are `1` to `20` and the columns reserved for sequence numbers are `15` to `25`, the margin will be from column `1` to `14`, and the columns reserved for sequence numbers will be from `15` to `25`.

For more information on the `#pragma sequence` directive, refer to "sequence" on page 174.

options

The C/VSE directive `#pragma options` is additional to the SAA Standard.

```
                        ┌──────,──┐
                        ▼         │
►►──#pragma──options──(───option──┴──)──►◄
```

The `#pragma options` directive specifies a list of compile-time options that are to be processed as if you had specified them in the PARM parameter of the JCL EXEC statement for the compile. The only compile-time options allowed on a `#pragma options` directive are:

AGGREGATE│NOAGGREGATE	CHECKOUT│NOCHECKOUT	GONUMBER│NOGONUMBER
HWOPTS│NOHWOPTS	INLINE│NOINLINE	NAME│NONAME
OPTIMIZE│NOOPTIMIZE	RENT│NORENT	SPILL
START	TEST│NOTEST	UPCONV│NOUPCONV
XREF│NOXREF		

For a detailed description of these options refer to the *C/VSE User's Guide*.

If you use a compile-time option that contradicts the options specified on the `#pragma options` directive, the compile-time option overrides the options on the `#pragma options` directive.

If you use one of the following compile-time options, the option name is inserted at the bottom of your object module indicating that it was used:

GONUMBER	INLINE	NAME
OPTIMIZE (both levels)	RENT	START
TARGET	TEST	UPCONV

page

```
►►──#pragma──page──(─────────)──►◄
                     └─n─┘
```

The `#pragma page` skips *n* pages of the source listing. If *n* is not specified, the compiler moves to the next page of the source listing.

pagesize

```
►►──#pragma──pagesize──(─────────)──►◄
                         └─n─┘
```

The `#pragma pagesize` directive sets the number of lines per page to *n* for the source listing.

C/VSE Addition to SAA Standard
The default page size is 60 lines. The minimum page size that you should set is 25.

runopts

The C/VSE directive `#pragma runopts` is additional to the SAA Standard.

```
                            ┌──┌─,─┐──┐
►►──#pragma──runopts──(──▼──option──┴──)──►◄
```

The `#pragma runopts` directive specifies a list of run-time options that are to be used at execution time. Specify your `#pragma runopts` directive in the source file that contains `main()`.

Refer to the *LE/VSE Programming Reference* for descriptions of specific run-time options.

sequence

The C/VSE directive `#pragma sequence` is additional to the SAA Standard.

```
►►──#pragma──┬──sequence──(──m──,──┬─n─┬──)──┬──►◄
             │                     └─*─┘     │
             └──nosequence───────────────────┘
```

The `#pragma sequence` directive specifies the section of the input record that is to contain sequence numbers. The `#pragma nosequence` directive specifies that the input record does not contain sequence numbers. The sequence setting specified by the `#pragma sequence` directive applies only to the file (source file or include file) that contains it, and has no effect on other `#include` files in the file. The sequence number area specified on the `#pragma sequence` directive comes into effect on the line following the directive and remains in effect until another `#pragma sequence` or a `#pragma nosequence` directive is encountered or until the end of the file is reached.

If you use the compile-time `SEQUENCE` or `NOSEQUENCE` option with the `#pragma sequence` or `#pragma nosequence` directives, the `#pragma` directive overrides the compile-time options. The compile-time option is in effect up to, and including, the `#pragma sequence` or `#pragma nosequence` directive. The default setting is `sequence(73,80)` for fixed-length records and `nosequence` for variable-length records.

In the syntax diagram the following parameters are specified:

m The column number of the left-hand margin. The value of *m* must be:

Greater than `0` and less than `32768`
and
Less than or equal to the value of *n*

n The column number of the right-hand margin. The value of *n* must be greater than `0` and less than `32768`.

An asterisk (*) can be assigned to *n* indicating the last column of the input record. Thus, `sequence(74,*)` indicates that sequence numbers are between column `74` and the end of the input record.

You can use `#pragma sequence` and `#pragma margins` together. If they reserve the same columns, `#pragma sequence` has priority and the columns will be reserved for sequence numbers. For example, if the columns reserved for margin are `1` to `20` and the columns reserved for sequence numbers are `15` to `25`, the margin will be from column `1` to `14`, and the columns reserved for sequence numbers will be from `15` to `25`. For more information on the `#pragma margins` directive, refer to "margins" on page 172.

skip

```
►►──#pragma──skip──(──┬───┬──)──►◄
                      └─n─┘
```

The `#pragma skip` skips the next *n* lines of the source listing. The value of *n* must be a positive integer less than 255. If *n* is omitted, one line is skipped.

strings

```
►►──#pragma──strings──(──┬─readonly─┬──)──►◄
                         └─writable─┘
```

The `#pragma strings` directive specifies that the compiler place strings into read-only memory or into writable memory. Strings are writable by default. This pragma must appear before any C code in a file.

C/VSE Addition to SAA Standard

`writable` is the default.

subtitle

```
►►──#pragma──subtitle──(──"subtitle"──)──►◄
```

The `#pragma subtitle` places the text specified by *subtitle* on all subsequent pages of the created source listing.

target

The C/VSE directive `#pragma target` is additional to the SAA Standard.

```
►►──#pragma──target──(──┬────┬──)──►◄
                        └─LE─┘
```

In some implementations of C, the `#pragma target` directive has several options for specifying the operating system or run-time environment for which the object module is to be created.

In C/VSE, the default behavior is to generate code to run with the LE/VSE run-time library. This is the same as specifying the `LE` option. There are no other options.

If you have more than one preprocessor directive, the only `#pragma`s that can precede `#pragma target` are `#pragma filetag`, `#pragma chars`, `#pragma langlvl`, and `#pragma longname`.

title

```
►►──#pragma──title──(──"title"──)──►◄
```

The `#pragma title` places the text specified by *title* on all subsequent pages of the source listing.

variable

The C/VSE directive `#pragma variable` is an addition to the SAA Standard.

```
►►──#pragma──variable──(──identifier──,──┬─RENT───┬──)──►◄
                                         └─NORENT─┘
```

The `#pragma variable` directive specifies that the named variable is to be used in a reentrant or non-reentrant fashion. Variables are reentrant when the `RENT` compile-time option is specified. If `NORENT` is specified and a `#pragma variable` is specified with `RENT`, writable static is forced. Refer to the *LE/VSE Programming Guide* for information on reentrancy.

Appendix A. Conforming to ANSI Standards

This appendix describes changes made to the C/VSE compiler and LE/VSE C run-time library for conformance to the *American National Standard for Information Systems - Programming Language C* standard. It also describes implementation-defined behavior of the C/VSE compiler which is not defined by ANSI.

Implementation-Defined Behavior

The following sections describe how C/VSE defines some of the implementation-defined behavior from the ANSI C Standard.

Identifiers

The number of significant characters in an identifier with no external linkage:

- 255

The number of significant characters in an identifier with external linkage:

- 255 with the compile-time option `LONGNAME` specified
- 8 otherwise

Case sensitivity of external identifiers:

- The VSE linkage editor truncates all external names to 8 uppercase characters so any external identifiers in upper and lower case map to the same name. However, the C/VSE compiler catches any such cases so that an identifier in both upper and lower case is flagged as an error to help you write portable code.

Characters

Source and execution characters which are not specified by the ANSI standard:

- The caret (^) character in ASCII (bitwise exclusive OR symbol) or the equivalent not (¬) character in EBCDIC.
- The vertical broken line (¦) character in ASCII which may be represented by the vertical line (|) character on EBCDIC systems.

Shift states used for the encoding of multibyte characters:

- The shift states are indicated with the `SHIFTIN` (hex value x0E) and the `SHIFTOUT` (hex value x0F) characters. Refer to the *LE/VSE C Run-Time Library Reference* for more information on wide character strings.

The number of bits that represent a character:

- 8 bits

The mapping of members of the source character set (characters and strings) to the execution character set:

- The same code page is used for the source and execution character set.

The value of an integer character constant that contains a character/escape sequence not represented in the basic execution character set:

- A warning is issued for an unknown character/escape sequence and the `char` is assigned the character following the back slash.

The value of a wide character constant that contains a character/escape sequence not represented in the extended execution character set:

- A warning is issued for the unknown character/escape sequence and the `wchar_t` is assigned the wide character following the back slash.

The value of an integer character constant that contains more than one character:

- The lowest four bytes represent the character constant.

The value of a wide character constant that contains more than one multibyte character:

- The lowest four bytes of the multibyte characters are converted to represent the wide character constant.

Equivalent type of `char`: signed char, unsigned char, or user-defined:

- The default for `char` is `unsigned`

Is each sequence of white-space characters (excluding the newline) retained or replaced by one space character?

- Any spaces or comments in your source program will be interpreted as one space.

String Conversion

Additional implementation-defined sequence forms that can be accepted by `strtod()`, `strtol()`, and `strtoul()` functions in other than the C locale:

- None

Integers

Table 17. Integers

Type	**Amount of Storage**	**Range (in `limits.h`)**
`signed short`	2 bytes	`-32768` to `32767`
`unsigned short`	2 bytes	`0` to `65535`
`signed int`	4 bytes	`-2147483647-1` to `2147483647`
`unsigned int`	4 bytes	`0` to `4294967295`
`signed long`	4 bytes	`-2147483647L-1` to `2147483647`
`unsigned long`	4 bytes	`0` to `4294967295`

The result of converting an integer to a `signed char`:

- The lowest 1 byte of the integer is used to represent the `char`. See the *LE/VSE C Run-Time Library Reference* for more information on data conversions.

The result of converting an integer to a shorter `signed` integer:

- The lowest 2 bytes of the integer are used to represent the `short int`.

The result of converting an `unsigned` integer to a `signed` integer of equal length, if the value cannot be represented:

- The bit pattern is preserved and the sign bit has no significance.

The result of bitwise operations (|, &, ^) on `signed int`:

- The representation is treated as a bit pattern and 2's complement arithmetic is performed.

The sign of the remainder of integer division if either operand is negative:

- The remainder is negative if exactly one operand is negative.

The result of a right shift of a negative-valued `signed` integral type:

- The result is sign extended and the sign is propagated.

Floating Point

Table 18. Floating Point

Type	Amount of Storage	Range of Exponents (base 10)
`float`	4 bytes	-78 to 75
`double`	8 bytes	-78 to 75
`long double`	16 bytes	-78 to 75

The direction of truncation when an integral number is converted to a floating-point that cannot exactly represent the original value:

- The value is truncated.

The direction of rounding when a floating-point number is converted to a narrower floating-point number:

- The floating-point number is truncated.

Arrays and Pointers

The type of `size_t`:

- `unsigned int`

The type of `ptrdiff_t`:

- `int`

The result of casting a pointer to an integer:

- The bit patterns are preserved.

The result of casting an integer to a pointer:

- The bit patterns are preserved.

Registers

How `register` storage class specifier affects the storage of objects in registers:

- If there is a register available, the object is stored in a register.

Structures, Unions, Enumerations, Bitfields

The result if a member of a union object is accessed using a member of a different type:

- The result is undefined.

The alignment/padding of structure members:

- If the structure is not packed, then padding is added to align the structure members on their natural boundaries. If the structure is packed, no padding is added. Refer to “C Data Mapping” on page 92 for more information on C data mapping.

The padding at the end of structure/union:

- Padding is added to end the structure on its natural boundary. The alignment of the `struct` or `union` is that of its strictest member. Refer to “C Data Mapping” on page 92 for more information on C data mapping.

The type of an `int` bit-field (`signed int`, `unsigned int`, user defined):

- The default is `unsigned`.

The order of allocation of bit-fields within an `int`:

- Bit-fields are allocated from low memory to high memory. For example, `0x12345678` would be stored with byte 0 containing `0x12`, and byte 3 containing `0x78`.

The rule for bit-fields crossing a storage unit boundary:

- Bit-fields can cross storage unit boundaries.

The integral type that represents the values of an enumeration type:

- Enumerations can have the type `char`, `short`, or `long` and be either `signed` or `unsigned` depending on their smallest and largest values.

Declarators

The maximum number of declarators (pointer, array, function) that can modify an arithmetic, structure, or union type:

- The only constraint is the availability of system resources.

Statements

The maximum number of `case` values in a `switch` statement:

- Because the `case` values must be integers and cannot be duplicated, the limit is `INT_MAX`.

Preprocessing Directives

Does the value of a single-character constant in a constant expression that controls conditional inclusion match the value of the character constant in the execution character set?

- Yes

Can such a constant have a negative value?

- Yes

The method of searching include source files (< >):

- See the *C/VSE User's Guide*.

Is the search for quoted source file names supported ("...")?

- User include files can be specified in double quotes. See the *C/VSE User's Guide*.

The mapping between the name specified in the include directive and the external source file name:

- See the *C/VSE User's Guide*.

The behavior of each pragma directive:

- See “#pragma” on page 162.

`__DATE__` and `__TIME__` is always defined to the date and time of compilation.

`__TIMESTAMP__` is always defined to the value:

```
"Mon Jan  1 01:01:01 1990"
```

(In some implementations of C, `__TIMESTAMP__` returns the date and time when the source file was last modified.)

Library Functions

The definition of `NULL` macro:

- `NULL` is defined to be a `((void *)0)`.

The format of diagnostic printed by the `assert` macro, and the termination behavior (abort behavior):

- When `assert` is executed, if the expression is false, the diagnostic message written by the `assert` macro has the format:

  ```
  Assertion failed:  expression, file filename, line line-number
  ```

Set of characters tested by the `is...()` functions:

- To create a table of the characters set up by the `ctype.h` functions use the program in the following example. The columns are organized by function as follows:

(Column 1)	The hexadecimal value of the character
AN	`isalnum()`
A	`isalpha()`
C	`iscntrl()`
D	`isdigit()`
G	`isgraph()`
L	`islower()`
(Column 8)	`isprint()`
PU	`ispunct()`
S	`isspace()`
PR	`isprint()`
U	`isupper()`
X	`isxdigit()`

EDCXRABG

```
 /* This example prints out ctest characters */

#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int ch;

   for (ch = 0; ch <= 0xff; ch++)
      {
      printf("%#04X ", ch);
      printf("%3s ", isalnum(ch)  ? "AN" : " ");
      printf("%2s ", isalpha(ch)  ? "A"  : " ");
      printf("%2s",  iscntrl(ch)  ? "C"  : " ");
      printf("%2s",  isdigit(ch)  ? "D"  : " ");
      printf("%2s",  isgraph(ch)  ? "G"  : " ");
      printf("%2s",  islower(ch)  ? "L"  : " ");
      printf("%c",  isprint(ch)  ? ch   : ' ');
      printf("%3s",  ispunct(ch)  ? "PU" : " ");
      printf("%2s",  isspace(ch)  ? "S"  : " ");
      printf("%3s",  isprint(ch)  ? "PR" : " ");
      printf("%2s",  isupper(ch)  ? "U"  : " ");
      printf("%2s",  isxdigit(ch) ? "X"  : " ");

      putchar('\n');
      }
}
```

The result of calling `fmod()` function with the second argument zero (return zero, domain error):

- `fmod()` returns a `0`.

Error Handling

The format of the message generated by the `perror()` and `strerror()` functions:

- See the *LE/VSE Debugging Guide and Run-Time Messages* for the messages emitted for `perror()` and `strerror()`.

  Note: `errno` is not emitted with the message.

How diagnostic messages are recognized:

- Refer to the *C/VSE User's Guide* and the *LE/VSE Debugging Guide and Run-Time Messages* for the lists of messages provided with C/VSE.

The different classes of messages:

- Messages are classified as shown by the following table.

Type of Message	Numeric Severity Level	Return Code
Information	00	0
Warning	10	4
Error	30	12
Severe error	> 30	16

How the level of diagnostic can be controlled:

- Use the compile-time option `FLAG` to control the level of diagnostic. There is also a compile-time option `CHECKOUT` which provides programming style diagnostics to aid you in determining possible programming errors.

Signals

The set of signals for the `signal()` function:

- See the *LE/VSE C Run-Time Programming Guide*.

The parameters and the usage of each signal recognized by the `signal()` function:

- See the *LE/VSE C Run-Time Programming Guide*.

The default handling and the handling at program start-up for each signal recognized by `signal()` function:

- `SIG_DFL` is the default signal. See the *LE/VSE C Run-Time Programming Guide* for more information on signal handling.

The signal blocking performed if the equivalent of `signal(sig, SIG_DFL)` is not executed at the beginning of signal handler:

- See the *LE/VSE C Run-Time Programming Guide*.

Is the default handling reset if a `SIGKILL` is received by a signal handler?

- Whenever you enter the signal handler, `SIG_DFL` becomes the default.

Translation Limits

System-determined means that the limit is determined by your system resources.

Table 19. Translation Limits

Nesting levels of:	
• Compound statements	• System-determined
• Iteration control	• System-determined
• Selection control	• System-determined
• Conditional inclusion	• System-determined
• Parenthesized declarators	• System-determined
• Parenthesized expression	• System-determined
Number of pointer, array and function declarators modifying an arithmetic a structure, a union, and incomplete type declaration	• System-determined
Significant initial characters in:	
• Internal identifiers	• 255
• Macro names	• 255
• External identifiers	• 8 (without `LONGNAME`)
Number of:	
• External identifiers in a translation unit	• System-determined
• Identifiers with block scope in one block	• System-determined
• Macro identifiers simultaneously declared in a translation unit	• System-determined
• Parameters in one function definition	• System-determined
• Arguments in a function call	• System-determined
• Parameters in a macro definition	• System-determined
• Parameters in a macro invocation	• System-determined
• Characters in a logical source line	• 32767
• Characters in a string literal	• 4K
• Bytes in an object	• `LONG_MAX` (see Note 1)
• Nested include files	• `SHRT_MAX`
• Enumeration constants in an enumeration	• System-determined
• Levels in nested structure or union	• System-determined

Note:

1. `LONG_MAX` is the limit for automatic variables only. For all other variables, the limit is 16MB.

Streams, Records, and Files

Does the last line of a text stream require a terminating newline character?

- No, the last newline character is defaulted.

Do space characters, that are written out to a text stream immediately before a newline character, appear when read?

- White space characters written to fixed record format text streams before a newline do not appear when read. However, white space characters written to variable record format text streams before a newline character appear when read.

The number of null characters that can be appended to the end of the binary stream:

- No limit

Where is the file position indicator of an append-mode stream initially positioned?

- The file position indicator is positioned at the end of the file.

Does a write on a text stream cause the associated file to be truncated?

- Yes

Does a file of zero length exist?

- Yes

The rules for composing a valid file name:

- See the *LE/VSE C Run-Time Programming Guide*.

Can the same file be simultaneously opened multiple times?

- For reading, the file can be opened multiple times; for writing/appending, the file can be opened once. Once a file is opened for update, it may be opened for reading. Once a file is opened for reading, it cannot be opened for writing.

The effect of the `remove()` function on an open file:

- `remove()` fails.

The effect of the `rename()` function on a file to a name that exists prior to the function call:

- `rename()` fails.

Are temporary files removed if the program terminates abnormally?

- For SAM files, no.
- For VSAM (including SAM ESDS), files are removed according to the DISP parameter on the DLBL statement. For details, see the *VSE/VSAM User's Guide*, SC33-6535 (VSE/ESA Version 1) or the *VSE/VSAM User's Guide and Application Programming*, SC33-6632 (VSE/ESA Version 2).

The effect of calling the `tmpnam()` function more than `TMP_MAX` times:

- `tmpnam()` fails and returns `NULL`.

The output of `%p` conversion in the `fprintf()` function:

- It is equivalent to `%X`.

The input of `%p` conversion in the `fscanf()` function:

- The value is treated as an integer.

The interpretation of a `-` character that is neither the first nor the last in the scanlist for `%[` conversion in the `fscanf()` function:

- The sequence of characters on either side of the - are used as delimiters. For example, `%[a-f]` will read in characters between `'a'` and `'f'`.

The value of `errno` on failure of `fgetpos()` and `ftell()` functions:

- This depends on the failure. For a list of the messages associated with `errno`, see the *LE/VSE Debugging Guide and Run-Time Messages*.

Memory Management

The behavior of `calloc()`, `malloc()`, and `realloc()` functions if the size requested is zero:

- Nothing is performed for `calloc()` and `malloc()`; `realloc()` frees the storage.

Environment

The arguments of the `main()` function:

- You can pass arguments to `main()` through `argv` and `argc`.

What happens with open files when the `abort()` function is called?

- The files are closed.

What is returned to the host environment when the `abort()` function is called?

- The return code of `2000` is returned.

The form of successful termination when the `exit()` function is called with argument zero or `EXIT_SUCCESS`:

- All files are closed, all storage is released and the return code of `0` is returned.

The form of unsuccessful termination when the `exit()` function is called with argument `EXIT_FAILURE`:

- All files are closed, all storage is released and the return code of `EXIT_FAILURE` is returned.

What status is returned by the `exit()` function if the argument is other than zero, `EXIT_FAILURE` and `EXIT_SUCCESS`?

- The argument of the `exit()` function is returned.

Environment variables:
Retrieving and setting environment variables using the `getenv()` and `setenv()` functions is described in the *LE/VSE C Run-Time Programming Guide*.

The format and a mode of execution of a string on a call to the `system()` function:

- See the *LE/VSE C Run-Time Library Reference*.

Localization

The environment specified by the `""` locale on a `setlocale()` call:

- `EDC$S370`

The supported locales:

- See the *LE/VSE C Run-Time Programming Guide*.

Time

The local time zone and Daylight Saving Time:

- This is specified in the locale.

The era for the `clock()` function:

- The era starts when the program is started by either a call from the operating system, or a call to `system()`. To measure the time spent in a program, call the `clock()` function at the start of the program, and subtract its return value from the value returned by subsequent calls to `clock()`.

Bibliography

IBM C for VSE/ESA Publications

Licensed Program Specifications, GC09-2421

Installation and Customization Guide, GC09-2422

Migration Guide, SC09-2423

User's Guide, SC09-2424

Language Reference, SC09-2425

Diagnosis Guide, GC09-2426

IBM Language Environment for VSE/ESA Publications

Fact Sheet, GC33-6679

Concepts Guide, GC33-6680

Debugging Guide and Run-Time Messages, SC33-6681

Installation and Customization Guide, SC33-6682

Licensed Program Specifications, GC33-6683

Programming Guide, SC33-6684

Programming Reference, SC33-6685

Run-Time Migration Guide, SC33-6687

Writing Interlanguage Communication Applications, SC33-6686

C Run-Time Programming Guide, SC33-6688

C Run-Time Library Reference, SC33-6689

Softcopy Publications

The following collection kit contains the C/VSE, LE/VSE, and other LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

You can order these publications from Mechanicsburg through your IBM representative.

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, SC20-1699.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

absolute value. The magnitude of a real number regardless of its algebraic sign.

abstract code unit (ACU). A measurement used by the C/VSE compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

ACU. Abstract code unit.

address. A name, label, or number identifying a location in storage, a device in a system or network, or any other data source.

aggregate. An array or a structure. Also, a compile-time option to show the layout of a structure or union in the listing.

alias. An alternate label used to refer to the same data element or point in a computer program.

alignment. See *boundary alignment*.

American National Standard Code for Information Interchange (ASCII). The code developed by ANSI for information interchange among data processing systems, data communications systems, and associated equipment. The ASCII character set consists of 7-bit control characters and symbolic characters.

Note: IBM has defined an extension to ASCII code (characters 128-255).

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

anonymous union. A union that is declared within a structure and that does not have a name.

ANSI. American National Standards Institute.

API. Application program interface.

application. The use to which an information processing system is put, for example, a payroll application, an airline reservation application, a network application.

application program interface (API). The formally defined programming language interface between an IBM system control program or a licensed program and the user of the program.

argument. In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called a parameter.

arithmetic object. An integral object, a bit field, or floating-point object.

array. A variable that contains an ordered group of data objects. All objects in an array have the same data type.

array element. A single data item in an array.

ASCII. American National Standard Code for Information Interchange.

assembly language. A symbolic programming language in which the set of instructions includes the instructions of the machine and whose data structures correspond directly to the storage and registers of the machine.

assignment conversion. A change to the form of the right operand that makes the right operand have the same data type as the left operand.

assignment expression. An operation that stores the value of the right operand in the storage location specified by the left operand.

associativity. The order for grouping operands with an operator (either left-to-right or right-to-left).

automatic calling. Calling in which the elements of the selection signal are entered into the data network contiguously at the full data signalling rate.

B

binary. (1) Pertaining to a system of numbers to the base two; the binary digits are 0 and 1. (2) Involving a choice of two conditions, such as on-off or yes-no.

binary expression. An expression containing two operands and one operator.

binary stream. An ordered sequence of untranslated characters.

bit field. A member of a structure or union that contains a specified number of bits.

block. The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

block statement. Any number of data definitions, declarations, and statements that appear between the symbols { and }. The block statement is considered to be a single C-language statement.

boundary alignment. The position in main storage of a fixed-length field (such as byte or doubleword) on an integral boundary for that unit of information. For example, on System/370 a word boundary is a storage address evenly divisible by two.

break statement. A language control statement that contains the word **break** and a semicolon. It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

buffer. A portion of storage used to hold input or output data temporarily.

buffer flush. A process that removes the contents of a buffer. After a buffer flush, the buffer is empty.

built-in. A function which the compiler will automatically inline instead of the function call unless the programmer specifies not to.

C

C language. A general-purpose high-level programming language.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions.

C language statement. A C language statement contains zero or more expressions. All C language statements, except block statements, end with a **;** (semicolon) symbol. A block statement begins with a **{** (left brace) symbol, ends with a **}** (right brace) symbol, and contains any number of statements.

C library. A system library that contains common C language subroutines for file access, memory allocation, and other functions.

call. To transfer control to a procedure, program, routine, or subroutine.

case clause. In a `switch` statement, a `case` label followed by any number of statements.

case label. The word `case` followed by a constant expression and a colon.

cast expression. An expression that converts the type of the operand to a specified scalar data type (the operator).

cast operator. The cast operator is used for explicit type conversions.

cataloged procedures. A set of control statements placed in a library and retrievable by name.

char specifier. A char is a built-in data type. In C, **char**, **signed char**, and **unsigned char** are all distinct data types.

character constant. A character or an escape sequence enclosed in single quotation marks.

character set. A group of characters used for a specific reason; for example, the set of characters a printer can print or a keyboard can support.

character variable. A data object whose value can be changed during program execution and whose data type is **char**, **signed char**, or **unsigned char.**

CICS. Customer Information Control System.

collating sequence. A specified arrangement for the order of characters in a character set.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

compile. To transform a set of programming language statements (source file) into machine instructions (object module).

compiler. A program that translates instructions written in a programming language (such as C language) into machine language.

complex number. A complex number is made up of two parts: a real part and an imaginary part. A complex number can be represented by an ordered pair (*a*, *b*), where *a* is the value of the real part and *b* is the value of the imaginary part. The same complex number could also be represented as *a* + *bi*, where *i* is the square root of -1.

conditional compilation statement. A preprocessor statement that causes the preprocessor to process specified source code in the file depending on the evaluation of a specific condition.

const. An attribute of a data object that declares the object cannot be changed.

constant expression. An expression having a value that can be determined during compilation and that does not change during program execution.

control statement. A statement that changes the path of execution.

conversion. A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values. Because accuracy of data representation varies among different data types, information may be lost in a conversion.

D

data object. A storage area used to hold a value.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.

data type. A category that specifies the interpretation of a data object such as its mathematical qualities and internal representation.

DBCS. (1) See *double-byte character set*. (2) See *ASCII*.

decimal constant. A numerical data type used in standard arithmetic operations.

declaration. A description that makes an external object or function available to a function or a block.

declare. To identify the variable symbols to be used at preassembly time.

default. An attribute, value or option that is used when no alternative is specified by the programmer.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default clause. In a **switch** statement, the keyword **default** followed by a colon, and one or more statements. When the conditions of the specified **case** labels in the **switch** statement do not hold, the **default** clause is chosen.

default initialization. The initial value assigned to a data object by the compiler if no initial value is specified by the programmer. **extern** and **static** variables receive a default initialization of zero, while the default initial value for **auto** and **register** variables is undefined.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. A data description that reserves storage and may provide an initial value.

demangling. The conversion of mangled names back to their original source code names. See also *mangling*.

denormal. Pertaining to a number with a value so close to `0` that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

digit. Any of the numerals from 0 through 9.

domain. All the possible input values for a function.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, entering, displaying, and printing DBCS characters requires hardware and supporting software that are DBCS capable.

double precision. Pertaining to the use of two computer words to represent a number with greater accuracy. For example, a floating-point number would be stored in the long format.

doubleword. A sequence of bits or characters that comprises two computer words and can be addressed as a unit.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time.

dynamic binding. Binding that occurs at run time.

E

EBCDIC. See *extended binary-coded decimal interchange code*.

E-format. Floating-point format, consisting of a number in scientific notation.

element. The component of an array, subrange, enumeration, or set.

enumeration constant. An identifier that is defined in an enumerator and that has an associated integer value. You can use an enumeration constant anywhere an integer constant is allowed.

enumeration data type. A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

enumeration tag. The identifier that names an enumeration data type.

enumerator. An enumeration constant and its associated value.

EOF. End of file.

escape sequence. A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: `a`, `b`, `f`, `n`, `r`, `t`, `v`, `'`, `"`, `x`, `\`, or followed by one to three octal or hexadecimal digits.

exception. In C, any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine.

executable program. A program that can be run on a processor.

expression. A representation for a value. For example, variables and constants appearing alone or in combination with operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 eight-bit characters.

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A definition appearing outside a function. The defined object is accessible to all functions that follow the definition and are located within the same source file as the definition.

F

fetch control block (FECB). An executable dynamic stub which is created by a `fetch()` function call. The stub transfers control to the true entry point of the module specified in the `fetch` call. The stub also switches the writable static environment thereby giving each instance of the `fetched` routine its own global data.

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

float constant. A constant representing a nonintegral number.

foreground processing. The execution of a computer program that preempts the use of computer facilities.

free store. Dynamically allocates memory. New and delete are used to allocate and deallocate free store.

function. A named group of statements that can be invoked and evaluated and can return a value to the calling statement.

function call. An expression that moves the path of execution from the current function to a specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of arguments.

function declarator. The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters.

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a ; (semicolon). It is required by the compiler when the function will be declared later so type checking can occur.

function scope. Labels that are declared in a function have function scope and can be used anywhere in that function.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

G

global. Pertaining to information available to more than one program or subroutine.

global scope. See *file scope*.

global variable. A variable defined in one portion of a computer program and used in at least one other portion of the computer program.

H

halfword. A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit.

hard error. An error condition on a network that requires that the network be reconfigured or that the source of the error be removed before the network can resume reliable operation.

header file. A file that contains system-defined control information that precedes user data.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. The special characters are `\x`, `0x`, or `0X`.

I

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

include file. A text file that contains declarations used by a group of functions, programs, or users. Also known as a header file.

initialize. To set the starting value of a data object.

initializer. An expression used to initialize data objects. In C, there are two types of initializers:

- An expression followed by an assignment operator is used to initialize fundamental data type objects.
- An expression enclosed in braces (**{}**) is used to initialize aggregates.

inlined function. Inlining is a hint to the compiler to perform inline expansion of the body of a function member. Functions declared and defined simultaneously in a class definition are inline. You can also explicitly declare a function inline by using the keyword **inline**. Both member and nonmember functions can be inlined. You can direct the compiler to inline a function with the `inline` keyword.

input stream. A sequence of control statements and data submitted to a system from an input unit.

instruction. A program statement that specifies an operation to be performed by the computer, along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

integer constant. A decimal, octal, or hexadecimal constant.

integral boundary. A location in main storage at which a fixed-length field, such as a halfword or doubleword, must be positioned. The address of an integral boundary is a multiple of the length of the field, expressed in bytes.

integral object. A character object, an object having an enumeration type, an object having variations of the type `int`, or an object that is a bit field.

internal data definition. A description of a variable appearing at the beginning of a block that causes storage to be allocated for the lifetime of the block.

interrupt. A temporary suspension of a process caused by an external event, performed in such a way that the process can be resumed.

intrinsic function. A function supplied by a program as opposed to a function supplied by the compiler.

IPL. Initial Program Load.

ISA. Initial Storage Area.

J

JCL. Job Control Language.

K

keyword. (1) A predefined word reserved for the C language, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

L

L-name. An external C name in an object module or an external non-C name in an object module produced by compiling with the LONGNAME option.

label. (1) An identifier followed by a colon. It is the target of a goto statement. (2) An identifier within or attached to a set of data elements.

labeled statement. A possibly empty statement immediately preceded by a label.

late binding. See *dynamic binding*.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, function calls, subroutines, or other data. (2) A set of object modules that can be specified in a link command.

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

linkage editor. Synonym for linker.

linker. A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single executable program (phase).

literal. See *constant*.

loader. A routine, commonly a computer program, that reads data into main storage.

local. Pertaining to information that is defined and available in only one function of a computer program.

local scope. A name declared in a block has local scope and can only be used in that block.

long constant. An integer constant followed by the letter L in uppercase or lowercase.

lvalue. An expression that represents a data object that can be both examined and altered.

M

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor `#define` directive.

main function. A function with the identifier `main` that is the first user function to get control when program execution begins. Each C program must have exactly one function named `main`.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. The linker uses these mangled names to ensure type-safe linkage.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

map. A set of values having a defined correspondence with the quantities or values of another set.

map file. A listing file that can be created during the link step and that contains information on the size and mapping of segments and symbols.

mapping. The establishing of correspondences between a given logical structure and a given physical structure.

mask. A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

member. A data object in a structure or a union.

metalanguage. A language used to specify another language.

migrate. To move to a changed operating environment, usually to a new release or version of a system.

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multiprocessing. Simultaneous or parallel processing of two or more computer programs or sequences by a multiprocessor.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of more than one task or program.

N

newline character. A control character that causes the print or display position to move to the first position on the next line. This control character is represented by `\n` in the C language.

NULL. A pointer guaranteed not to point to a data object.

null character (\0). The ASCII or EBCDIC character with the hex value `00`, all bits turned off.

null value. A parameter position for which no value is specified.

O

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as C language).

object module. A portion of an object program produced by a compiler from a source program, and suitable as input to a linkage editor.

octal. A base eight numbering system.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

operand. An entity on which an operation is performed.

operating system. Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

operation. A specific action such as add, multiply, shift.

operator. A symbol (such as `+`, `-`, `*`) that represents an operation (in this case, addition, subtraction, multiplication).

overflow. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overflow condition. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overlay. To write over existing data in storage.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C operators when the functions and operators are used with class types.

P

pack. To store data in a compact form in such a way that the original form can be recovered.

pad. To fill unused positions in a field with data, usually zeros, ones, or blanks.

parameter declaration. A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

persistent environment. A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

pointer. A variable that holds the address of a data object or function.

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

precision. A measure of the ability to distinguish between nearly equal values.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that are then executed, resulting in the alteration of the source program.

preprocessor statement. A statement that begins with the symbol `#` and is interpreted by the preprocessor.

primary expression. An identifier, a parenthesized expression, a function call, an array element specification, or a structure or union member specification.

process. An instance of an executing application and the resources it uses.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters.

R

record. The unit of data transmitted to and from a program.

recoverable error. An error condition that allows continued execution of a program.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

register. A storage area commonly associated with fast-access storage, capable of storing a specified amount of data such as a bit or an address.

reserved word. In programming languages, a keyword that may not be used as an identifier.

rounding. To omit one or more of the least significant digits in a positional representation and to adjust the remaining digits according to a specified rule. The purpose of rounding is usually to limit the precision of a number or to reduce the number of characters in the number.

run-time library. A collection of functions in object code form, whose members can be referred to by an application program during the linking step.

S

S-name. An external non-C name in an object module produced by compiling with the NOLONGNAME option. Such a name is up to 8 characters long and single case.

SAA. Systems Application Architecture.

scalar. An arithmetic object, or a pointer to an object of any type.

scope. That part of a source program in which an object is defined and recognized.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape.

signal. A condition that may or may not be reported during program execution. For example, `SIGFPE` is the signal used to represent erroneous arithmetic operations such as a division by zero.

signal handler. A function to be called when the signal is reported.

single-byte character set. A set of characters in which each character is represented by 1 byte of storage.

single precision. Pertaining to the use of one computer word to represent a number, in accordance with the required precision.

software signal. A signal that is explicitly raised by the user (by using the `raise` function).

source file. A file that contains source statements for such items as language programs and data description specifications.

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

SQL. Structured Query Language.

stack. An area of storage used for keeping variables associated with each call to a function or block.

stand-alone. Pertaining to operation that is independent of any other device, program, or system.

statement. An instruction that ends with the character **;** (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

static binding. Binding that occurs at compilation time based on the resolution of overloaded functions.

storage class specifier. One of: **auto**, **register**, **static**, or **extern**.

stream. See *data stream*.

string constant. Zero or more characters enclosed in double quotation marks.

structure. A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

structure tag. The identifier that names a structure data type.

stub routine. Within run-time libraries, contains the minimum lines of code required to locate a given routine at run time.

subsystem. A secondary or subordinate system, or programming support, usually capable of operating independently of or asynchronously with a controlling system.

swap. To exchange one thing for another.

switch expression. The controlling expression of a **switch** statement.

switch statement. A C language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile.

Systems Application Architecture (SAA). Pertaining to the definition of a common programming interface, conventions, and protocols for designing and developing applications with cross-system consistency.

T

tag. One or more characters attached to a set of data that identifies the set.

task. One or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer.

template function. A function generated by a function template.

thread. A unit of execution within a process.

trap. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred.

trigraph sequence. A combination of three keystrokes used to represent unavailable characters in a C source program. Before preprocessing, each trigraph sequence in a string or a literal is replaced by the single character that it represents.

truncate. To shorten a value to a specified length.

try block. A block in which a known C exception is passed to a handler.

type. The description of the data and the operations that can be performed on or by the data. Also see *data type*.

type balancing. A conversion that makes both operands have the same data type. If the operands do not have the same size data type, the compiler converts the value of the operand with the smaller type to a value having the larger type.

type conversion. See *boundary alignment*.

type definition. A definition of a data type.

type specifier. Used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unary expression. An expression that contains one operand.

underflow. A condition that occurs when the result of an operation is less than the smallest possible nonzero number.

union. A construct that can hold any one of several data types, but only one data type at a time.

union tag. The identifier that names a union data type.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

V

variable. An object that can take different values at different times.

visible. Visibility of identifiers is based on scoping rules and is independent of *access.*

volatile. An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

VSAM. Virtual Storage Access Method.

W

whitespace. Space characters, tab characters, form feed characters, and newline characters.

wide character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

word boundary. The storage position at which data must be aligned for certain processing operations. The halfword boundary must be divisible by 2, the fullword boundary by 4, and the doubleword boundary by 8.

Z

zero suppression. The removal of, or substitution of blanks for, leading zeros in a number. For example, `00057` becomes `57` when using zero suppression.

Index

Communicating Your Comments to IBM

IBM C for VSE/ESA
Language Reference
Release 1

Publication No. SC09-2425-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@vnet.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

IBM C for VSE/ESA
Language Reference
Release 1

Publication No. SC09-2425-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	☐	☐	☐	☐	☐

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
SC09-2425-00

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK   ONTARIO   CANADA   M3C 1H7

Fold and Tape | **Please do not staple** | Fold and Tape

SC09-2425-00

Cut or Fold
Along Line

Program Number: 5686-A01

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC09-2425-00