Rational® ClearCase®

Building Software

VERSION: 2003.06.00 AND LATER

PART NUMBER: 800-026157-000

UNIX EDITION

Rational®
the software development company

support@rational.com
http://www.rational.com

Legal Notices

Copyright ©1992-2003, Rational Software Corporation. All Rights Reserved.
Part Number: 800-026157-000
Version Number: 2003.06.00

This manual (the "Work") is protected under the copyright laws of the United States and/or other jurisdictions, as well as various international treaties. Any reproduction or distribution of the Work is expressly prohibited without the prior written consent of Rational Software Corporation.

The Work is furnished under a license and may be used or copied only in accordance with the terms of that license. Unless specifically allowed under the license, this manual or copies of it may not be provided or otherwise made available to any other person. No title to or ownership of the manual is transferred. Read the license agreement for complete terms.

Rational Software Corporation, Rational, Rational Suite, Rational Suite ContentStudio, Rational Apex, Rational Process Workbench, Rational Rose, Rational Summit, Rational Unified process, Rational Visual Test, AnalystStudio, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, PerformanceStudio, PureCoverage, Purify, Quantify, Requisite, RequisitePro, RUP, SiteCheck, SiteLoad, SoDa, TestFactory, TestFoundation, TestMate and TestStudio are registered trademarks of Rational Software Corporation in the United States and are trademarks or registered trademarks in other countries. The Rational logo, Connexis, ObjecTime, Rational Developer Network, RDN, ScriptAssure, and XDE, among others, are trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. Government Restricted Rights

Licensee agrees that this software and/or documentation is delivered as "commercial computer software," a "commercial item," or as "restricted computer software," as those terms are defined in DFARS 252.227, DFARS 252.211, FAR 2.101, OR FAR 52.227, (or any successor provisions thereto), whichever is applicable. The use, duplication, and disclosure of the software and/or documentation shall be subject to the terms and conditions set forth in the applicable Rational Software Corporation license agreement as provided in DFARS 227.7202, subsection (c) of FAR 52.227-19, or FAR 52.227-14, (or any successor provisions thereto), whichever is applicable.

Warranty Disclaimer

This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements

Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

Microsoft, the Microsoft logo, Active Accessibility, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, bCentral, BizTalk, Bookshelf, ClearType, CodeView, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectX, DirectXJ, DoubleSpace, DriveSpace, FrontPage, Funstone, Genuine Microsoft Products logo, IntelliEye, the IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, Mapbase, MapManager, MapPoint, MapVision, Microsoft Agent logo, the Microsoft eMbedded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, NetMeeting, NetShow, the Office logo, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, RelayOne, Rushmore, SharePoint, SourceSafe, TipWizard, V-Chat, VideoFlash, Visual Basic, the Visual Basic logo, Visual C++, Visual C#, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX, are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or in other countries.

Sun, Sun Microsystems, the Sun Logo, Ultra, AnswerBook 2, medialib, OpenBoot, Solaris, Java, Java 3D, ShowMe TV, SunForum, SunVTS, SunFDDI, StarOffice, and SunPCi, among others, are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

Licensee shall not incorporate any GLOBEtrotter software (FLEXlm libraries and utilities) into any product or application the primary purpose of which is software license management.

BasicScript is a registered trademark of Summit Software, Inc.

Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Copyright © 1995 by Addison-Wesley Publishing Company, Inc. All rights reserved.

Copyright ©1997 OpenLink Software, Inc. All rights reserved.

This software and documentation is based in part on BSD Networking Software Release 2, licensed from the Regents of the University of California. We acknowledge the role of the Computer Systems Research Group and the Electrical Engineering and Computer Sciences Department of the University of California at Berkeley and the Other Contributors in its development.

This product includes software developed by Greg Stein <gstein@lyra.org> for use in the mod_dav module for Apache (http://www.webdav.org/mod_dav/).

Additional legal notices are described in the legal_information.html file that is included in your Rational software installation.

Contents

Preface xv
About This Manual xv
ClearCase Documentation Roadmap xvi
Typographical Conventions. xvii
Online Documentation xviii
Customer Support xix

ClearCase Build Concepts 1
Overview of the ClearCase Build Scheme 1
View Context Required 3
How Builds Work 3
Build Reference Time and Build Sessions 3
Exit Status 4
Dependency Tracking of MVFS and Non-MVFS Files 4
Automatic Detection of MVFS Dependencies 4
Tracking Non-MVFS Files 5
Derived Objects and Configuration Records. 5
Build Avoidance 5
Hierarchical Builds 7
Automatic Dependency Detection 7
Express Builds 7
Build Auditing with clearaudit 8
Compatibility with Other make Programs 8
Parallel Building 9
The Parallel Build Procedure 9
Building on a Non-ClearCase Host 10

Derived Objects and Configuration Records 11
Derived Objects Overview. 11
Derived Object Naming 12
Configuration Records 13
Configuration Record Example. 13
Contents of a Configuration Record 15
Header Section 15

MVFS Objects Section 16
Non-MVFS Objects Section 16
Variables and Options Section 16
Build Script Section 16
Configuration Record Hierarchies 16
Configuration Record Cache 19
Kinds of Derived Objects 20
Shareable DOs 20
Nonshareable DOs 20
Storage of Derived Objects 21
Promotion and Winkin 21
DO Versions 24
Reuse of DO IDs 24
Derived Object Reference Counts 25

Pointers on Using ClearCase Build Tools 27

Invoking clearmake 27
A Simple clearmake Build Scenario 27
Accommodating Build Avoidance 30
Increasing the Verbosity Level of a Build 30
Handling Temporary Changes in the Build Procedure 30
Specifying Build Options 31
Handling Targets Built in Multiple Ways 31
Using a Recursive Invocation of clearmake 32
Optimizing Winkin by Avoiding Pseudotargets 32
Accommodating the Build Tool's Different Name 32
Declaring Source Dependencies in Makefiles 33
Source Dependencies Declared Explicitly 34
Explicit Dependencies on Searched-For Sources 35
Build-Order Dependencies 36
Problems with Forced Builds 36
How clearmake Interprets Double-Colon Rules 36
Continuing to Work During a Build 37
Using Config Spec Time Rules 38
Inappropriate Use of Time Rules 39
Build Sessions, Subsessions, and Hierarchical Builds 39
Subsessions 40
Versions Created During a Build Session 40

Coordinating Reference Times of Several Builds 40
Objects Written at More Than One Level 40
Build Auditing and Background Processes 41
Working with Incremental Update Tools 42
Example: Building an Archive 42
Makefile Restructuring for Incremental Archive Targets 43
A Note on the Use of ar Keys 45
Example: Incremental Linking 45
Additional Incremental-Update Situations 45
Adding a Version String or Time Stamp to an Executable 46
Creating a what String 46
Implementing a –Ver Option 47

Working with DOs and Configuration Records 49

Setting Correct Permissions for Derived Objects 49
Listing and Describing Derived Objects 50
Listing Derived Objects Created at a Certain Pathname 50
Listing a Derived Object's Kind 50
Displaying a DO's OID 51
Displaying a Description of a DO Version 51
Identifying the Views That Reference a Derived Object 52
Caching Unavailable Views 52
Specifying Views That Can Wink In Derived Objects 53
Specifying a Derived Object in Commands 53
Winking In a DO Manually 54
Preventing Winkin 54
Preventing Winkin to Your View 54
Preventing Winkin to Other Views 55
Using Express Builds to Prevent Winkin to Other Views 55
Enabling Express Builds 55
Configuring an Existing View for Express Builds 56
Creating a New View That Uses Express Builds 56
Preventing Winkin to or from Other Architectures 56
Converting Derived Objects to View-Private Files 56
Working with DO Versions 57
Creating DO Versions 57
Checking In DOs During a Build 57
Accessing DO Versions 58

Displaying Configuration Records for DO Versions 58
DOs in Unavailable Views 61
Releasing DOs 61
Converting Nonshareable DOs to Shared DOs 62
Automatic Conversion of Nonshareable DOs to Shareable DOs 62
Creating Links to Derived Objects 63
Displaying VOB Disk Space Used for Derived Objects 64
Deleting Derived Objects 64
Removing Data Containers for Derived Objects 64
Scrubbing Derived Objects and Data Containers 64
Degenerate Derived Objects 65
Data Container Deleted 65
DO Deleted from VOB Database 65
CR Unavailable 65
Displaying Contents of Configuration Records 66
Comparing Configuration Records 66
Attaching Labels or Attributes to Versions in a CR 66
Configuring a View to Select Versions Used to Build a DO 66
Including a Makefile Version in a Configuration Record 67

clearmake Makefiles and BOS Files 69

Makefile Overview 69
Build Options Specification Files 70
Format of Makefiles 71
Restrictions 72
Libraries 72
Command Echoing and Error Handling 72
Built-In Rules 73
Include Files 73
Macros 73
Order of Precedence of Make Macros and Environment Variables 73
Make Macros 74
Internal Macros 75
VPATH Macro 76
Special Targets 77
Special Targets for Use in Makefiles 77
Special Targets for Use in Makefiles or BOS Files 78
Sharing Makefiles Between UNIX and Windows 83

BOS File Entries 83
Standard Macro Definitions 83
Target-Dependent Macro Definitions 84
Shell Command Macro Definitions 84
Special Targets 84
Include Directives 84
Comments 84
Conflict Resolution 85
SHELL Environment Variable 86
CCASE_BRANCH0_REUSE Environment Variable 86

Using clearmake Compatibility Modes 87

Using ClearCase to Build C++ Programs 89

Working with Templates 90
Explicit Instantiation 90
Alternative to Using the Procedures in This Chapter 91
Precompiled Header Files 91
Working with Cfront-Based C++ Compilers 91
Cfront Template Instantiation: Interaction with clearmake 92
Link-Time Cfront Template Instantiation 93
How Link-Time Instantiation Interferes with clearmake 94
Models for Working with Cfront-Based Compilers 95
The Simple Model 95
How the Simple Model Works 96
Sample Scenario Using the Simple Model 96
Limitations of the Simple Model 97
The Multiple Repositories Model 98
Using a Recognized Compiler Macro 98
Inserting Special Build Rules in Your Makefile 99
Using an Alternate (CM-Safe) Multiple Repository Model 100
Example Makefile Using the Multiple Repository Model 101
Testing the Makefile 101
How the Multiple Repositories Model Works 102
Limitations of the Multiple Repositories Model 103
The Forced Instantiation Model 104
Maintaining Dummy Source Files 105
Setting Up the Makefile 106
How the Forced Instantiation Model Works 107

Limitations of the Forced Instantiation Model. 107
Working with SPARCompiler C++. 108
SPARCompiler Template Instantiation: Interaction with clearmake 108
Setting Up the Repository. 109
Cleaning the Repository 109
Models for Working with SPARCompiler C++ 110
The Simple Model. 110
How the Simple Model Works 110
Sample Scenario Using the Simple Model. 111
Limitations of the Simple Model 111
Building Archives That Contain Template Code 112
Managing Template References 112
Building the Archive 113
The Multiple Repositories Model 113
Using a Recognized Compiler Macro. 114
Inserting Special Build Rules in Your Makefile. 114
Example Makefile Using the Multiple Repositories Model 115
Testing the Makefile. 116
How the Multiple Repositories Model Works 116
Limitations of the Multiple Repositories Model. 117
Building Archives That Contain Template Code 117
Multiple Repositories Example. 118
Working with the SGI Delta/C++ Compiler 119
SGI Delta/C++ Compiler Template Instantiation: Interaction with clearmake. . 120
Automatic Instantiation. 120
Compile-Time Demand Instantiation 120
Explicit Instantiation 121
Working with the IBM AIX XLC C++ Compiler 122
XLC Compiler Template Instantiation: Interaction with clearmake 122
Models for Working with IBM XLC 123
The Simple Model. 123
Modifying the Source Files. 123
Designing Your Makefile 124
Limitations of the Simple Model 124
The Compile-Time Demand Instantiation Model 125
Modifying the Source Files. 125
Designing Your Makefile 126
Duplicate Symbol Warnings from the Linker 126
The Explicit Instantiation Model 126
Modifying the Source Files. 127

Designing Your Makefile 127
Working with the HP aC++ Compiler 127
Automatic Instantiation 127
Command-Line Option Instantiation 128
Explicit Instantiation 128

Using ClearCase Build Tools with Java 131

Using make Tools with javac 131
Using javac with clearmake 131
ClearCase Build Problems with javac 132
Using the clearmake makefile Special Target 132
Unsupported Builds 133
Makefile Requirements for .JAVAC 133
Using the javaclasses Built-in Macro 134
Deriving Class Dependencies 135
Storing Class Dependencies 136
Java Cyclic Class Dependencies 136
Using .class.dep Files 136
Dependencies and DO Reuse/Winkin 136
Effects of Nested Classes 137
.JAVAC in BOS Files 137
.SIBLINGS_AFFECT_REUSE 137
Building Java Applications Successfully without .JAVAC 137
Writing Correct Makefiles 138
No Mutually Dependent Files 138
Mutually Dependent Files 139
Allowing Rebuilds 139
Configuring Makefiles to Behave Like make 140

Setting Up a Parallel Build 141

Overview of Parallel Building 141
Parallel Build Scheduler 142
Failure Modes 143
Setting Up the Client Host 143
Creating Build Hosts Files 144
Load Balancing 146
Randomizing Host Selection 146
Idleness Threshold 147

Include File Facility 148
Including Comments in a File 148
Examples 148
Setting Up Trust Relationships 149
Setting Up Server Hosts 150
Examples 153
Starting a Parallel Build 153
Setting CCASE_HOST_TYPE in a Shell Startup Script 154
Preventing Parallel Builds of Targets 155
Preventing Exponential Invocations of abe 156

Building Software for Multiple Platforms 157

Issues in Multiple Platform Development 157
Handling Source Code Differences 157
Handling Build Procedure Differences 158
Alternative Approach Using imake 159
Segregating the Derived Objects of Different Variants 160
Approach 1: Use Architecture-Specific Subdirectories 160
Approach 2: Use Different Views 161
Multiple Architecture Example 161
Scenario 161
Defining Architecture-Specific CPP Macros 162
Creating Makefiles in the Source and Build Directories 162

Setting Up a Build on a Non-ClearCase Host 165

Build Scenario 165
Setting Up an Export View 165
Mounting the VOB Through the Export View 166
Revising the Build Script 166
Performing an Audited Build in the Export View 168

Index 169

Figures

Figure 1 Building Software with ClearCase: Isolation and Sharing 2
Figure 2 Parallel Building 9
Figure 3 Extended Pathname of a Derived Object 12
Figure 4 Kinds of Information in a Configuration Record 14
Figure 5 Configuration Record Hierarchy 17
Figure 6 Storage of a Shareable Derived Object 23
Figure 7 clearmake Build Scenario 29

Preface

Rational ClearCase is a comprehensive software version control and configuration management system.

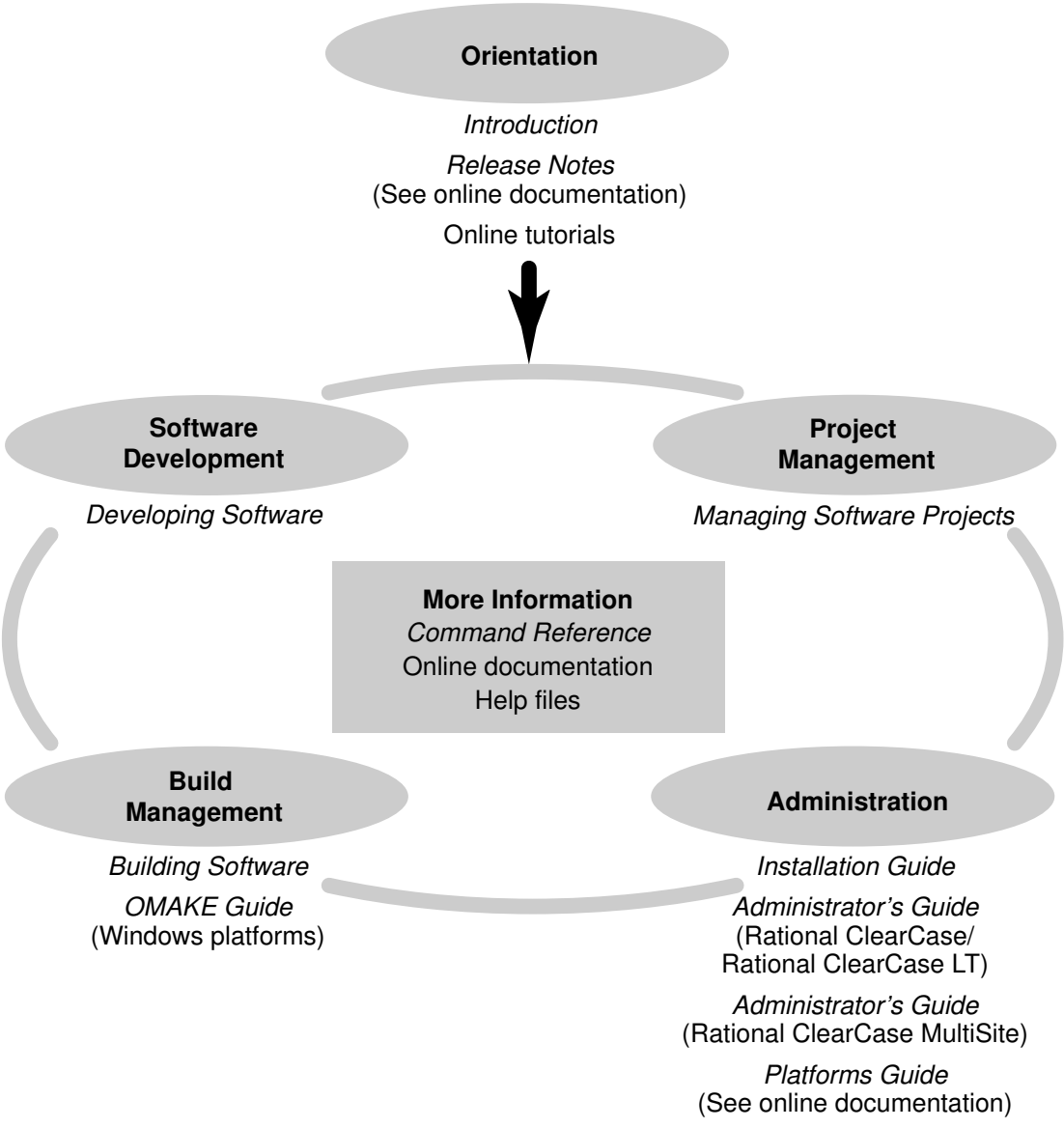
About This Manual

This manual provides an overview of ClearCase build management features and describes how to use ClearCase build tools. It is for new or experienced users of ClearCase who are familiar with software build concepts.

If you are not familiar with ClearCase build concepts and tools, read Chapter 1, *ClearCase Build Concepts*, Chapter 2, *Derived Objects and Configuration Records*, and Chapter 3, *Pointers on Using ClearCase Build Tools*.

For information about using ClearCase build tools with C++ programs or with Java tools, read Chapter 7, *Using ClearCase to Build C++ Programs* or Chapter 8, *Using ClearCase Build Tools with Java*.

ClearCase Documentation Roadmap



Typographical Conventions

This manual uses the following typographical conventions:

- *ccase-home-dir* represents the directory into which the ClearCase Product Family has been installed. By default, this directory is /opt/rational/clearcase on UNIX and C:\Program Files\Rational\ClearCase on Windows.
- *cquest-home-dir* represents the directory into which Rational ClearQuest has been installed. By default, this directory is /opt/rational/clearquest on UNIX and C:\Program Files\Rational\ClearQuest on Windows.
- **Bold** is used for names the user can enter; for example, command names and branch names.
- A sans-serif font is used for file names, directory names, and file extensions.
- **A sans-serif bold font** is used for GUI elements; for example, menu names and names of check boxes.
- *Italic* is used for variables, document titles, glossary terms, and emphasis.
- `A monospaced font` is used for examples. Where user input needs to be distinguished from program output, **bold** is used for user input.
- Nonprinting characters appear as follows: <EOF>, <NL>.
- Key names and key combinations are capitalized and appear as follows: SHIFT, CTRL+G.
- [ ] Brackets enclose optional items in format and syntax descriptions.
- { } Braces enclose a list from which you must choose an item in format and syntax descriptions.
- | A vertical bar separates items in a list of choices.
- ... In a syntax description, an ellipsis indicates you can repeat the preceding item or line one or more times. Otherwise, it can indicate omitted information.

  Note: In certain contexts, you can use "**...**" within a pathname as a wildcard, similar to "*" or "?". For more information, see the **wildcards_ccase** reference page.
- If a command or option name has a short form, a "medial dot" (·) character indicates the shortest legal abbreviation. For example:

  lsc·heckout

Online Documentation

The ClearCase Product Family (CPF) includes online documentation, as follows:

Help System: Use the **Help** menu, the **Help** button, or the F1 key. To display the contents of the online documentation set, do one of the following:

- On UNIX, type **cleartool man contents**
- On Windows, click **Start > Programs > Rational Software > Rational ClearCase > Help**
- On either platform, to display contents for Rational ClearCase MultiSite, type **multitool man contents**
- Use the **Help** button in a dialog box to display information about that dialog box or press F1.

Reference Pages: Use the **cleartool man** and **multitool man** commands. For more information, see the **man** reference page.

Command Syntax: Use the **–help** command option or the **cleartool help** command.

Tutorial: Provides a step-by-step tour of important features of the product. To start the tutorial, do one of the following:

- On UNIX, type **cleartool man tutorial**
- On Windows, click **Start > Programs > Rational Software > Rational ClearCase > ClearCase Tutorial**

PDF Manuals: Navigate to:

- On UNIX, *ccase-home-dir*/doc/books
- On Windows, *ccase-home-dir*\doc\books

Customer Support

If you have any problems with the software or documentation, please contact Rational Customer Support by telephone, fax, or electronic mail as described below. For information regarding support hours, languages spoken, or other support information, click the **Support** link on the Rational Web site at **www.rational.com**.

Your location	**Telephone**	**Facsimile**	**Electronic mail**
North America	800-433-5444 toll free or 408-863-4000 Cupertino, CA	408-863-4194 Cupertino, CA 781-676-2460 Lexington, MA	**support@rational.com**
Europe, Middle East, and Africa	+31-(0)20-4546-200 Netherlands	+31-(0)20-4546-201 Netherlands	**support@europe.rational.com**
Asia Pacific	61-2-9419-0111 Australia	61-2-9419-0123 Australia	**support@apac.rational.com**

ClearCase Build Concepts

1

Rational ClearCase supports makefile-based building of software systems and provides a software build environment closely resembling that of the **make** program. **make** was developed for UNIX systems and has been ported to other operating systems. You can use files controlled by ClearCase to build software, and use native **make** programs, third-party build utilities, your company's own build programs, or the ClearCase build tools **clearmake** and **clearaudit**.

The **clearmake** build tool provides compatibility with other **make** variants, along with powerful enhancements:

- Build auditing, with automatic detection of source dependencies, including header file dependencies
- Automatic creation of permanent bill-of-materials documentation of the build process and its results
- Sophisticated build-avoidance algorithms to guarantee correct results when building in a parallel development environment
- Sharing of binaries among views, saving both time and disk storage
- Parallel building, applying the resources of multiple processors and/or multiple hosts to builds of large software systems

The **clearaudit** build tool provides build auditing and creation of bill-of-materials documentation.

clearmake and **clearaudit** are intended for use in dynamic views. You can use them in a snapshot view, but the features that distinguish them from ordinary **make** programs (build avoidance, build auditing, derived object sharing, and so on) are not enabled in snapshot views; therefore, these features are not available when using **clearmake** with Rational ClearCase LT.

Overview of the ClearCase Build Scheme

Developers perform builds, along with all other work related to ClearCase, in views. Typically, developers work in separate, private views. Sometimes, a team shares a single view (for example, during a software integration period).

As described in *Developing Software*, each view provides a complete environment for building software that includes a particular configuration of source versions and a private work area in which you can modify source files, and use build tools to create object modules, executables, and so on.

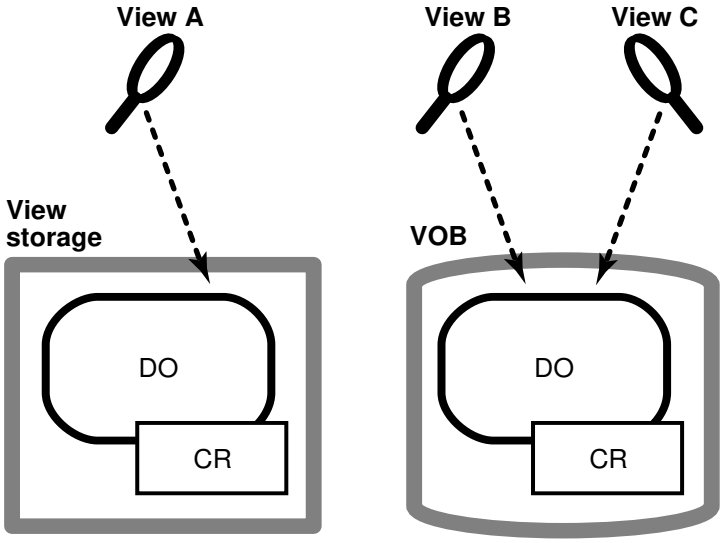
As a build environment, each view is partially isolated from other views. Building software in one view never disturbs the work in another view, even another build of the same program at the same time. However, when working in a dynamic view, you can examine and benefit from work done previously in another dynamic view. A new build shares files created by previous builds, when appropriate. This sharing saves the time and disk space involved in building new objects that duplicate existing ones.

You can (but need not) determine what other builds have taken place in a directory, across all dynamic views. ClearCase includes tools for listing and comparing past builds.

The key to this scheme is that the project team's VOBs constitute a globally accessible repository for files created by builds, in the same way that they provide a repository for the source files that go into builds. A file produced by a software build is a derived object (DO). Associated with each derived object is a configuration record (CR), which **clearmake** uses during subsequent builds to determine whether the DO can be reused or shared.

Figure 1 illustrates the ClearCase software build scheme.

Figure 1 Building Software with ClearCase: Isolation and Sharing



The section *Dependency Tracking of MVFS and Non-MVFS Files* on page 4 describes how ClearCase keeps track of the objects produced by software builds. *Build Avoidance* on page 5 describes the mechanism that enables such objects to be shared among views.

View Context Required

For a build that uses the data in one or more VOBs, the shell or command interpreter from which you invoke **clearmake** must have a view context. On UNIX systems, the view context must be either a set view or a working directory view. If you have a working directory view that differs from the set view, **clearmake** changes its set view to the working directory view.

You can build objects in a standard directory, without a view context, but doing so disables many of **clearmake**'s special features.

How Builds Work

In many ways, ClearCase builds adhere closely to the standard **make** paradigm:

1. You invoke **clearmake**, optionally specifying the names of one or more targets. (Such explicitly specified targets are called *goal targets*.)
2. **clearmake** reads zero or more makefiles, each of which contains targets and their associated build scripts. It also reads zero or more build options specification (BOS) files, which supplement the information in the makefiles.
3. **clearmake** supplements the makefile-based software build instructions with its own built-in rules or, when it runs in a compatibility mode, with built-in rules specific to that mode.
4. For each target, **clearmake** performs build avoidance, determining whether it actually needs to execute the associated build script (target rebuild). It takes into account both source dependencies (Have any changes occurred in source files used in building the target?) and build dependencies (Must other targets be updated before this one?).
5. If a target meets **clearmake**'s rebuild criteria, **clearmake** executes its build script.

Build Reference Time and Build Sessions

As your build progresses, other developers can continue to work on their files and may check in new versions of elements that your build uses. If your build takes an hour to complete, you do not want build scripts executed early in the build to use version 6 of a header file and scripts executed later to use version 7 or 8. To prevent such inconsistencies, **clearmake** locks out any version that meets both of these conditions:

- The version is selected by a configuration specification rule that includes the **LATEST** version label.
- The version was checked in after the time the build began (the build reference time).

This reference-time facility applies to checked-in versions of elements only; it does not lock out changes to checked-out versions, other view-private files, and non-MVFS objects. **clearmake** adjusts for time discrepancies between system clocks on different hosts in a network (clock skew).

Exit Status

clearmake returns a zero exit status if all goal targets are successfully processed. It returns a nonzero exit status in two cases:

- **clearmake** itself detects an error, such as a syntax error in the makefile. In this case, the error message includes the string `clearmake`.
- A makefile build script terminates with a nonzero exit status (for example, a compiler error).

Dependency Tracking of MVFS and Non-MVFS Files

During build-script execution in a dynamic view, a host's MVFS (multiversion file system) audits low-level system calls performed on ClearCase data: **create**, **open**, **read**, and so on. Calls involving the following objects are monitored:

- Versions of elements used as build input
- View-private files used as build input (for example, the checked-out version of a file element)
- Files created within VOB directories during the build

Some of these objects are stored in the VOB, and others are view-private files. The view combines them into a virtual work area, where they appear to be located in VOB directories. They are called *MVFS files* because they are accessed through the MVFS.

Automatic Detection of MVFS Dependencies

Because auditing of MVFS files is completely automated, you do not have to keep track of which files are being used in builds. ClearCase does the tracking instead. For example, it determines which C-language source files referenced with **#include** directives are used. Tracking eliminates the need both to declare such files in the makefile and for dependency-detection tools, such as **makedepend**.

If you store your build tools (compilers, linkers, and so on) as ClearCase elements and run them from the VOB, they are recorded in the configuration record as implicit detected dependencies.

Tracking Non-MVFS Files

A build can also involve files that are not accessed through VOB directories. Such non-MVFS files are not audited automatically, but are tracked if you declare them as dependencies in a makefile. This tracking enables auditing of build tools that are not stored as ClearCase elements (for example, a C-language compiler), flag files in the user's home directory, and so on. Tracking information on a non-MVFS file includes its absolute path, time stamp, size, and checksum.

Derived Objects and Configuration Records

When it finishes executing a build script, **clearmake** records the results, including build audit information, in the form of derived objects and configuration records.

A derived object (DO) is a file created in a VOB during a build or build audit with **clearmake**. Each DO has an associated configuration record (CR), which is the bill of materials for the DO. The CR documents aspects of the build environment, the assembly procedure for a DO, and all the files involved in the creation of the DO.

Note: All derived objects created by executing a build script have equal status, even though some of them may be explicit build targets, and others may be created as side effects of the build script (for example, compiler listing files). The term *siblings* describes a group of DOs created by the same script and associated with a single CR.

For more information about DOs and CRs, see Chapter 2, *Derived Objects and Configuration Records*.

Build Avoidance

clearmake attempts to avoid rebuilding derived objects. If an appropriate derived object exists in the view, **clearmake** reuses that DO. If there is no appropriate DO in the view, **clearmake** looks for an existing DO built in another view that can be winked in to the current view. The search process is called *shopping*.

The process of qualifying a candidate DO is called *configuration lookup*. It involves matching information in the VOB from the candidate DO's config record against the user's current *build configuration*. This process guarantees correct results in a parallel development environment, which the standard time-stamp-based algorithm used by

make cannot do. Even if an object module is newer than a particular version of its source file, the module may have been built using a different version. In fact, reusing object modules and executables built recently is likely to be incorrect when rebuilding a previous release of an application from old sources. The configuration lookup algorithm that ClearCase uses guarantees that your builds are both correct (inappropriate objects are not reused) and optimal (appropriate objects are always reused).

For a DO to be reused or winked in, the build configuration documented in its configuration record must match the current view's build configuration. The build configuration consists of two items:

- **Files**. The versions of elements listed in the CR must match the versions selected by the view in which the build is performed. Any view-private files or non-MVFS files listed in the CR must also match.
- **Build procedure**. The build options in the CR must match the build options specified on the command line, in the environment, in makefiles, or in build options specification files. The build script listed in the CR must match the script that will be executed if the target is rebuilt. The scripts are compared with all make macros expanded; thus, a match occurs only if the same build options apply (for example, "compile for debugging").

The search ends when **clearmake** finds a DO whose configuration matches the view's current build configuration exactly. In general, a configuration lookup can have three outcomes:

- **Reuse**. If the DO (and its siblings) in the view match the build configuration, **clearmake** keeps them.
- **Winkin**. If a DO built previously matches the build configuration, **clearmake** causes that DO and its siblings to appear in this view. This operation is called *winkin*.

  Note: clearmake does not contact all views to determine whether they contain DOs that can be winked in. Instead, it uses DO information in the VOB to eliminate inappropriate candidates. Only if it finds a candidate does it contact the containing view to retrieve the DO's configuration record.
- **Rebuild**. If configuration lookup fails to find a DO that matches the build configuration, **clearmake** executes the target's build script, which creates one or more new DOs and a new CR.

Reuse and winkin take place only if **clearmake** determines that a newly built derived object would be identical to the existing one. Winkin takes place when two or more views select the same versions of source elements used in a build. For example, you can create another view that has the same configuration as an existing view. Initially, the

new view sees all the sources but contains no derived objects. Running **clearmake** winks in many derived objects from the existing view.

Hierarchical Builds

In a hierarchical build, some objects are built and then used to build others. **clearmake** performs configuration lookup separately for each target. To ensure a consistent result, **clearmake** also applies this principle: When a new object is created, all targets that depend on it are rebuilt. Note that winkin does not cause rebuilds of dependencies.

Automatic Dependency Detection

Configuration records enable automatic checking of source dependencies as part of build avoidance. All such dependencies (for example, on C-language header files) are logged in a build's configuration record, whether or not they are explicitly declared in a makefile.

Express Builds

During an audited build, **clearmake** writes to the VOB information about a newly built DO. Configuration lookup by future builds uses that information to determine whether the DO is a candidate for winkin.

There is a performance tradeoff when you create DOs. While the build is writing the DO information to the VOB database, other users cannot write to the VOB. This performance loss is offset when the DO is used by subsequent builds, which can make those builds faster. But if the DO is never used by another view, the performance loss is not offset.

ClearCase express builds create derived objects, but do not write information to the VOB. These DOs are nonshareable and are not considered for winkin by other views. They can be reused only by the view in which they were built.

Express builds offer two advantages over regular builds:

- Scalability. During an express build, write access to the VOB is not blocked by time-consuming DO write operations. More users can build in a VOB without making VOB access slower.
- Performance. Express builds are faster than regular builds, because the build does not write DO information into the VOB.

Which kind of build occurs when you invoke **clearmake** depends on how your view is configured. To use express builds, you must use a dynamic view whose DO property

is set to nonshareable. For information about enabling express builds, see *Using Express Builds to Prevent Winkin to Other Views* on page 55.

Build Auditing with clearaudit

Some organizations, or some developers, may want to use ClearCase build auditing without using the **clearmake** program. Others may want to audit development activities that do not involve makefiles. These users can do their work in an audited shell, which is a standard shell with build auditing enabled.

For example, a technical writer produces formatted manual page files by running a shell script that invokes **nroff(1)**. When the script is executed in a dynamic view in a shell created by **clearaudit**, ClearCase creates a single configuration record, recording all the source versions used. All MVFS files read during execution of the audited shell are listed as inputs to the build. All MVFS files created become derived objects, associated with the single configuration record.

For more information, see the **clearaudit** reference page.

Compatibility with Other make Programs

Many **make** utilities are available in the multiple-architecture, multiple-vendor world of open systems. The ClearCase **clearmake** program shares features with many of them and has some unique features.

You can adjust the level of compatibility that **clearmake** has with other **make** programs:

- Suppress special features of **clearmake**.

  Use command options to turn off such features as winkin, comparison of build scripts, comparison of detected dependencies, and creation of DOs and CRs. You can turn off configuration lookup altogether, so that the standard time-stamp-based algorithm is used for build avoidance.

- Enable features of other **make** programs.

  clearmake has several compatibility modes, which provide for partial emulations of popular **make** programs. For more information, see the makefile compatibility commands in the *Command Reference*.

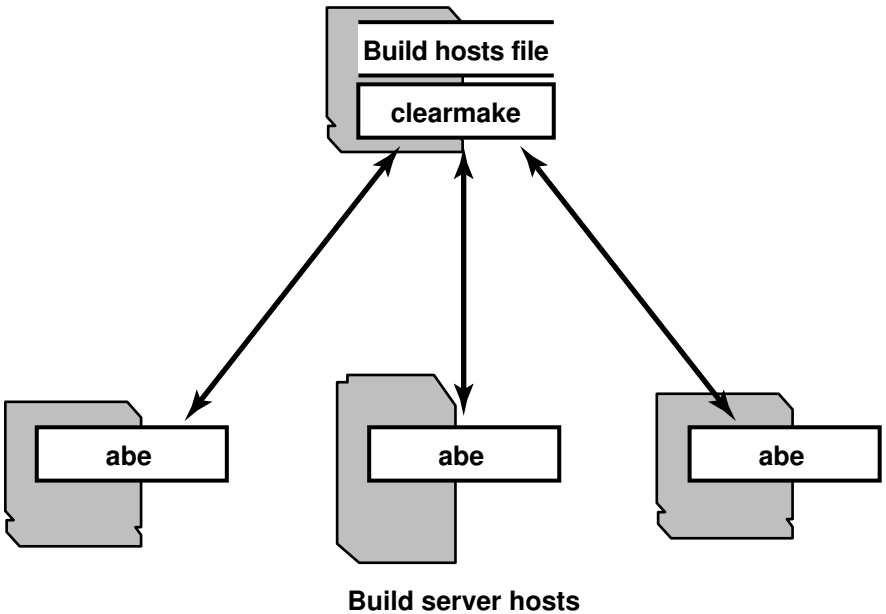
To achieve absolute compatibility with other **make** programs, you can actually use them to perform builds. However, builds with a standard **make** do not provide build auditing, configuration lookup, or sharing of DOs. The MVFS files that the build

creates are view-private files, not derived objects. To perform an audited build, you can execute the **make** program in a **clearaudit** shell.

Parallel Building

clearmake includes support for parallel building (concurrent execution of a set of build scripts on one or most hosts). A command option specifies the number of hosts to use; host names to use are read from a build hosts file (Figure 2).

Figure 2 Parallel Building



For example, you can perform a three-way build, all of whose processes execute on a single multiprocessor server; an overnight build can be distributed across all the workstations in the local network.

The Parallel Build Procedure

Before starting a build, **clearmake** parses the makefile to analyze the hierarchy of intermediate targets needed to build the final target. In a serial build, **clearmake** constructs each target (or reuses an existing DO) before continuing the analysis of subsequent builds.

In a parallel build, when **clearmake** detects that a target is out of date, it dispatches the build script for that target to one of the hosts listed in the build hosts file. **clearmake** continues to analyze the build hierarchy to detect other targets that can be built at the same time. It dispatches the build script for subsequent targets to one of the hosts in the build hosts file. The total number of build scripts being executed at any particular time is equal to or less than the number you specify with **–J**. Each target is built as soon as system resources on the build host allow.

Execution of build scripts is managed on each remote host by the ClearCase audited build executor (**abe**), which is invoked through standard remote-shell facilities.

For more information about parallel builds, see Chapter 9, *Setting Up a Parallel Build*.

Building on a Non-ClearCase Host

Many organizations develop multiple variants of their products, targeted at different platforms. If ClearCase is not available for some of the platforms, you can still build all the required product variants by using either cross-development or non-ClearCase access:

- Cross-development allows you to perform a build on a supported host to produce executables for the unsupported host.
- Non-ClearCase access allows hosts on which ClearCase is not installed to access ClearCase data, using standard network file-sharing services (such as NFS).

Non-ClearCase access takes advantage of view transparency. Through automatic version-selection, a view makes any VOB appear to be a standard directory tree. Any such VOB image can be exported to a non-ClearCase host.

Developers on the non-ClearCase host can perform builds within these VOB image directory trees, using native **make** utilities or other local build tools. The object modules and executables produced by such builds are stored in the view used to export the VOB.

For more information about building on a non-ClearCase host, see Chapter 11, *Setting Up a Build on a Non-ClearCase Host*.

Derived Objects and Configuration Records

2

This chapter describes derived objects and configuration records. Rational ClearCase creates derived objects and configuration records only if you build in a dynamic view with one of the ClearCase build tools. For information about managing derived objects and configuration records, see Chapter 4.

Derived Objects Overview

As described in Chapter 1, derived objects are created during builds with ClearCase build tools. They are used for build avoidance and derived object sharing.

In a parallel-development environment, it is likely that many DOs with the same pathname will exist at the same time. For example, suppose that source file msg.c is being developed on three branches concurrently, in three different views. ClearCase builds performed in those three views produce object modules named msg.o. Each of these is a DO, and each has the same standard pathname, for example, /vobs/proj/src/msg.o.

Note: Symbolic links created by a build script and files created in non-VOB directories are not DOs.

In addition, each DO can be accessed with ClearCase extended names:

- Within each dynamic view, a standard UNIX pathname accesses the DO referenced by that view. This is another example of the ClearCase transparency feature.

  `msg.o` *(the DO in the current view)*

- You can use a view-extended pathname to access a DO in any view:

  `/view/drp/vobs/proj/src/msg.o` *(the DO in view drp)*

  `/view/2_integ/vobs/proj/src/msg.o` *(the DO in view R2_integ)*

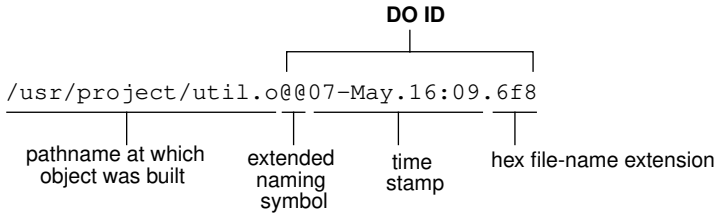
Derived Object Naming

No name collisions occur among derived objects built at the same pathname, because each DO is cataloged in the VOB database with a unique identifier, its *DO ID*. The DO ID references a DO independently of views. The **lsdo** command can list all DOs created at a specified pathname, regardless of which views (if any) can select them:

```
% cleartool lsdo hello.o
07-May.16:09   akp   "hello.o@@07-May.16:09.623" on neptune
06-May.12:47   akp   "hello.o@@06-May.12:47.539" on neptune
01-May.21:49   akp   "hello.o@@01-May.21:49.282" on neptune
03-Apr.21:40   akp   "hello.o@@01-May.21:40.226" on neptune
```

Together, a DO's standard name (hello.o) and its DO ID (**07-May.16:09.623**) constitute a VOB-extended pathname to that particular derived object (Figure 3). (The extended naming symbol is host specific; most organizations use the default value, **@@**.)

Figure 3 Extended Pathname of a Derived Object



Standard software must access a DO through a dynamic view, using a standard pathname or view-extended pathname. You can use such names with debuggers, profilers, **rm**, **tar**, and so on. Only ClearCase programs can reference a DO using a VOB-extended pathname, and only the DO's metadata is accessible in this way. You can use a view-extended pathname with the **winkin** command, to make the file system data of any DO available to your view. See *Winking In a DO Manually* on page 54.

The following example describes a DO with an extended pathname (hello@@07-Mar.11:40.217) and its configuration record:

```
% cleartool describe hello@@07-Mar.11:40.217
created 07-Mar-03.11:40.217 by Allison K. Pak (akp.users@phobos)
references: 1 => cobalt:/usr1/tmp/akp/tut/old.vws
```

```
% cleartool catcr hello@@07-Mar.11:40.217
Target hello built by akp.user
Host "cobalt" running SunOS 5.7 (sun4u)
Reference Time 07-Mar-03.11:40:41, this audit started
 07-Mar-03.11:40:46
View was cobalt:/var/tmp/akp/tut/old.vws
Initial working directory was /vobs/akp_cobalt_hw/src
----------------------------
MVFS objects:
----------------------------
/vobs/akp_cobalt_hw/src/hello@@07-Mar.11:40.217
/vobs/akp_cobalt_hw/src/hello.o@@07-Mar.11:40.213
/vobs/akp_cobalt_hw/src/util.o@@07-Mar.11:40.215
----------------------------
Variables and Options:
----------------------------
MKTUT_CC=cc
----------------------------
Build Script:
----------------------------
  cc -o hello hello.o util.o
----------------------------
```

```
% ls hello.@@07-Mar.11:40.217
Cannot access hello@@07-Mar.11:40.217: No such file or directory.
```

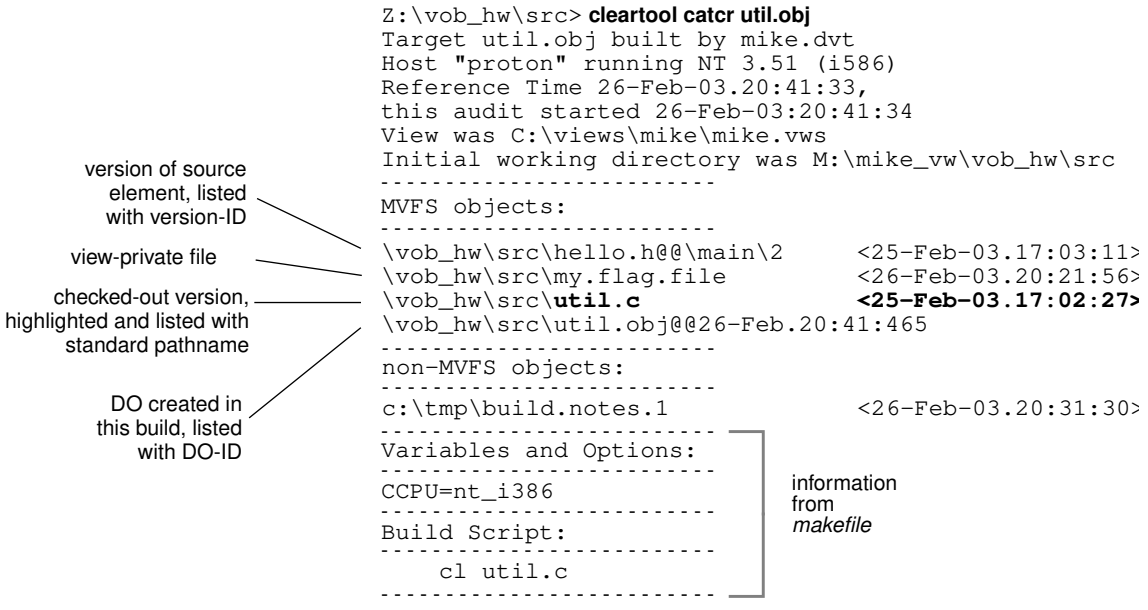
Configuration Records

A configuration record (CR) is the bill of materials for a derived object or set of DOs. The CR documents aspects of the build environment, the assembly procedure for a DO, and all the files involved in the creation of the DO.

Configuration Record Example

The **catcr** command displays the configuration record of a specified DO. Figure 4 shows a CR, with annotations to indicate the various kinds of information in the listing.

Figure 4 Kinds of Information in a Configuration Record




```
Z:\vob_hw\src> cleartool catcr util.obj
Target util.obj built by mike.dvt
Host "proton" running NT 3.51 (i586)
Reference Time 26-Feb-03.20:41:33,
this audit started 26-Feb-03:20:41:34
View was C:\views\mike\mike.vws
Initial working directory was M:\mike_vw\vob_hw\src
-------------------------
MVFS objects:
-------------------------
\vob_hw\src\hello.h@@\main\2     <25-Feb-03.17:03:11>
\vob_hw\src\my.flag.file         <26-Feb-03.20:21:56>
\vob_hw\src\util.c               <25-Feb-03.17:02:27>
\vob_hw\src\util.obj@@26-Feb.20:41:465
-------------------------
non-MVFS objects:
-------------------------
c:\tmp\build.notes.1             <26-Feb-03.20:31:30>
-------------------------
Variables and Options:
-------------------------
CCPU=nt_i386
-------------------------
Build Script:
-------------------------
    cl util.c
-------------------------
```

Some notes on Figure 4:

- 03Version of source element, listed with version-ID. By default, **catcr** does not list versions of the VOB directories involved in a build. To list this information, use the **–long** option:

  cleartool catcr –long util.o

  ```
  directory version /vobs/hw/.@@/main/1     <25-Feb-03.16:59:31>
  directory version /vobs/hw/src@@/main/3   <26-Feb-03.20:53:07>
  ...
  ```

- Declared dependencies. One principal feature of ClearCase is the automatic detection of source dependencies on MVFS files: versions of elements and objects in view-private storage. In addition, a CR includes non-MVFS objects that are explicitly declared as dependencies in the makefile. Figure 4 shows one such declared dependency, on file build.notes.1, located in the non-VOB directory /tmp.

- Listing of checked-out versions. Checked-out versions of file elements are highlighted. Checked-out versions of directory elements are listed like this:

  ```
  directory version /vobs/hw/src@@/main/CHECKEDOUT
  <26-Feb-03.17:05:23>
  ```

  When the elements are subsequently checked in, a listing of the same configuration record shows the updated information. For example,

  ```
  /vobs/hw/src/util.c                  <25-Feb-03.17:02:27>
  ```

becomes

```
/vobs/hw/src/util.c@@/main/4        <25-Feb-03.17:02:27>
```

The actual configuration record contains a ClearCase internal identifier for each MVFS object. After the version is checked in, **catcr** lists that object differently.

Notes:

- The time stamps in the configuration record are for informational purposes; they are not used for rebuild or winkin decisions. ClearCase uses OIDs to track versions used in builds.
- It is possible to produce two configuration records that show different time stamps for the same version of an object. If an object is checked out before a build, **clearmake** records the checkout time as the most recent modification time of the object. If you then check in the object and rebuild, **clearmake** records the checkin time as the most recent modification time of the object. Comparing the configuration records from both builds then shows that the same version of the object has different time stamps.

Contents of a Configuration Record

The following sections describe the contents of configuration records.

Header Section

As displayed by **catcr**, the header section of a CR includes the following lines:

- Makefile target associated with the build script and the user who started the build:

  ```
  Target util.o built by akp.dvt
  ```

  For a CR produced by **clearaudit**, the target is `ClearAudit_Shell`.

- Host on which the build script was executed, along with information from the **uname(2)** system call:

  ```
  Host 'neon' running SunOS 5.5.1 (sun4m)
  ```

- Reference time of the build (the time **clearmake** or **clearaudit** began execution), and the time when the build script for this particular CR began execution:

  ```
  Reference Time 15-Sep-03.08:18:56, this audit started
  15-Sep-03.08:19:00
  ```

  In a hierarchical build, involving execution of multiple build scripts, all the resulting CRs share the same reference time. (For more about reference time, see the **clearmake** reference page.)

- View storage directory of the view in which the build took place:

```
View was neptune:/home/akp/views/930825.vws
```

- Working directory at the time build script execution or **clearaudit** execution began:

```
Initial working directory was /vobs/hw/src
```

MVFS Objects Section

An *MVFS object* is a file or directory in a VOB. The MVFS Objects section of a CR includes this information:

- Each MVFS file or directory read during the build. These include versions of elements and view-private files used as build input, checked-out versions of file elements, DOs read, and any tools or scripts used during the build that are under version control.
- Each derived object produced by the target rebuild.

Non-MVFS Objects Section

A non-MVFS object is an object that is accessed outside a VOB (compiler, system-supplied header file, temporary file, and so on). The Non-MVFS Objects section of a CR includes each non-MVFS file that appears as an explicit dependency in the makefile or is a dependency inferred from a suffix rule. See *Declaring Source Dependencies in Makefiles* on page 33.

This section is omitted if there are no such files or if the CR was produced by **clearaudit**.

Variables and Options Section

The Variables and Options section of a CR lists the values of make macros referenced by the build script.

This section is omitted from a CR produced by **clearaudit**.

Build Script Section

The Build Script section of a CR lists the script that was read from a makefile and executed by **clearmake**.

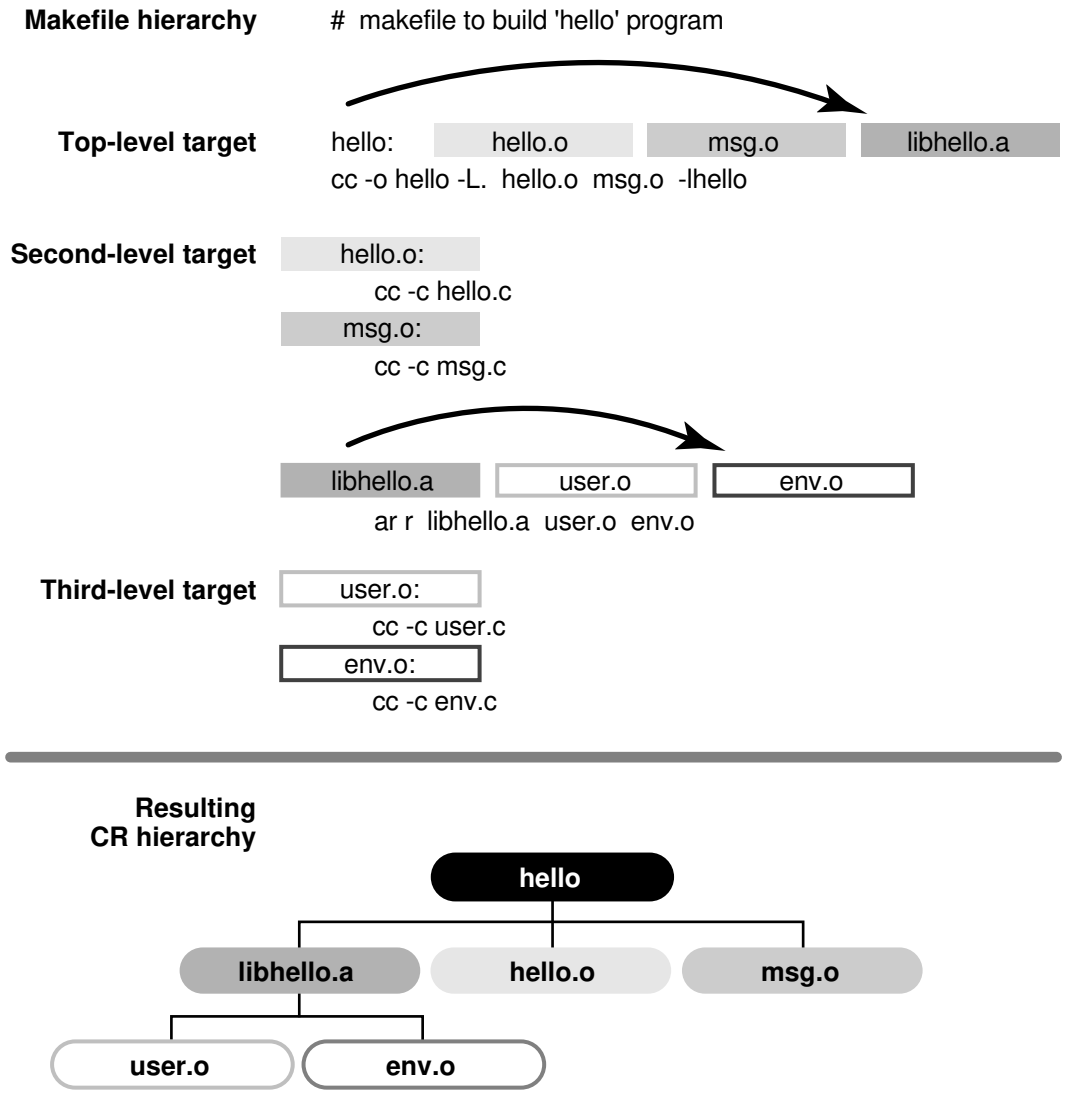
This section is omitted from a CR produced by **clearaudit**.

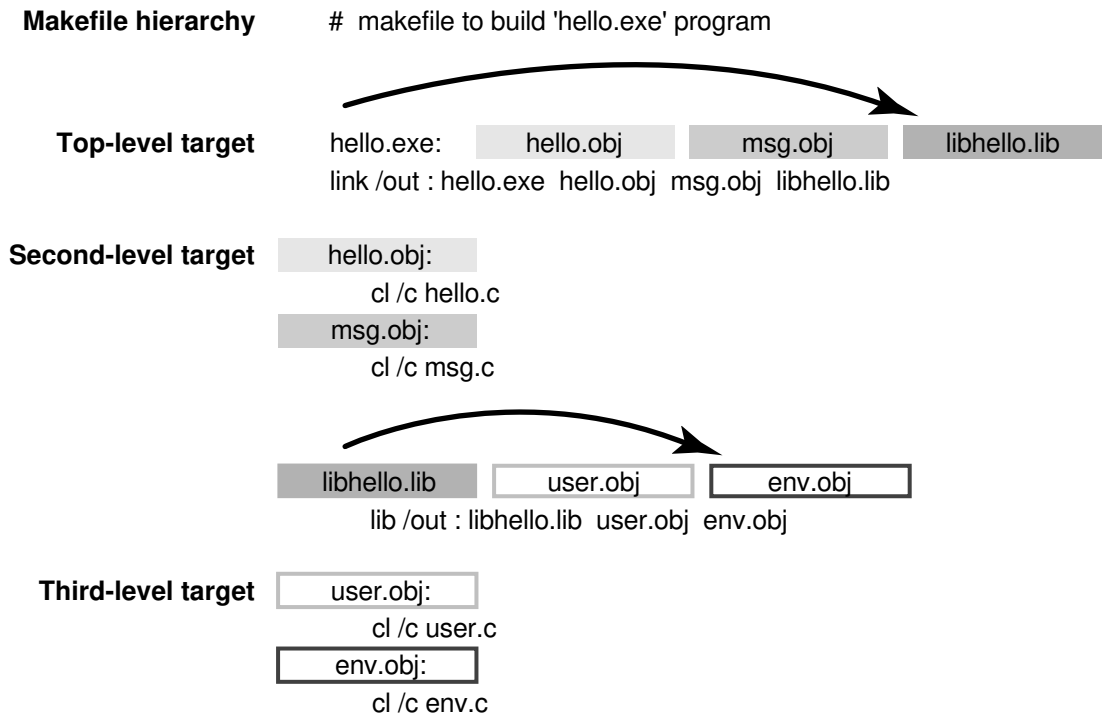
Configuration Record Hierarchies

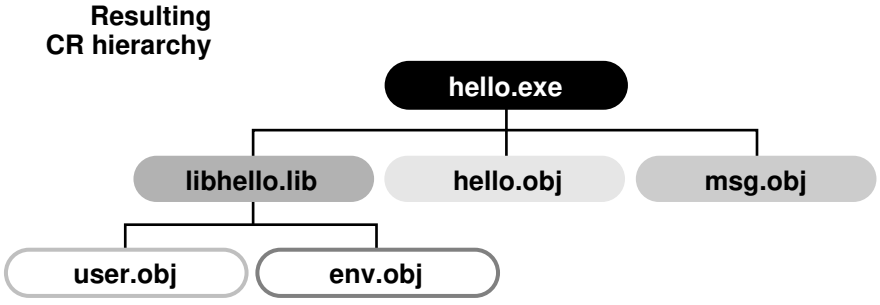
A typical makefile has a hierarchical structure. Thus, running **clearmake** once to build a high-level target can cause multiple build scripts to be executed and multiple CRs to

be created. Such a set of CRs can form a configuration record hierarchy, which reflects the structure of the makefile (Figure 5).

Figure 5 Configuration Record Hierarchy







An individual parent/child link in a CR hierarchy is established in one of two ways:

- In a target/dependencies line

  For example, the following target/dependencies line declares derived objects hello.o, msg.o, and libhello.a as build dependencies of derived object hello:

  ```
  hello: hello.o msg.o libhello.a
  ...
  ```

Accordingly, the CR for hello is the parent of the CRs for the .o files and the .a file.

- In a build script

  For example, in the following build script, derived object libhello.a in another directory is referenced in the build script for derived object hello:

  ```
  hello: $(OBJS)
       cd ../lib ; $(MAKE) libhello.a
       cc –o hello $(OBJS) ../lib/libhello.a
  ```

  Accordingly, the CR for hello is the parent of the CR for libhello.a.

  Note: The recursive invocation of **clearmake** in the first line of this build script produces a separate CR hierarchy, which is not necessarily linked to the CR for hello. The second line of the build script links the CR for ../lib/libhello.a with that of hello.

The **catcr** and **diffcr** commands have options for handling CR hierarchies:

- By default, they process individual CRs.
- With the **–recurse** option, they process the entire CR hierarchy of each derived object specified, keeping the individual CRs separate.
- With the **–flat** option, they combine (or flatten) the CR hierarchy of each specified derived object.

Some ClearCase features process entire CR hierarchies automatically. For example, when the **mklabel** command attaches version labels to all versions used to build a particular derived object (**mklabel –config**), it uses the entire CR hierarchy of the specified DO. Similarly, ClearCase maintenance procedures do not scrub the CR associated with a deleted DO if it is a member of the CR hierarchy of a higher-level DO.

Configuration Record Cache

When a derived object is created in a view, both its data container and its associated configuration record are stored in the view's private storage area. The CR is stored in the view database, in compressed format. To speed configuration lookup during subsequent builds in this view, a compressed copy of the CR is also cached in a view-private file, .cmake.state, located in the directory that was current when the build started.

When a DO is winked in for the first time, the associated CR moves from the view's private storage area to the VOB database, as shown in Figure 6.

Kinds of Derived Objects

During a regular build, ClearCase build tools create shareable derived objects. During an express build, they create nonshareable derived objects. Both kinds of DOs have configuration records, but only shareable DOs can be winked in by other views.

The following sections describe the kinds of DOs and their life cycles.

Shareable DOs

When a ClearCase build tool creates a shareable DO, it creates a configuration record for the DO and writes information about the DO into the VOB. (At this point, the DO is shareable but unshared.) Builds in other views use this information during configuration lookup. If the build determines that it can wink in an existing DO, it contacts the view containing the DO and promotes the DO to the VOB. (The DO is now shareable and shared.)

As noted in *Express Builds* on page 7, you must consider whether the performance benefit of winking in DOs is worth the performance cost of making them available for winkin.

Note: The process of looking for a DO to wink in uses an efficient algorithm to eliminate mismatches. The build tool does not contact other views to retrieve configuration records unless the configuration lookup process determines that there is a winkin candidate.

The configuration lookup process cannot guarantee that the DO is suitable for use. The process uses details in the config record to determine whether a DO is suitable for winkin, but the config record does not record all parameters of a build. For example, a config record may list only a compiler's name and the options used. If two builds use incompatible compilers with the same name, unwanted winkins from one build to the other can occur.

Note: To minimize occurrences of incorrect winkin, all developers must use the same set of tools. An effective way to do so is to put your build tools under version control and always run them from the VOB.

Nonshareable DOs

During an express build, the ClearCase build tool creates nonshareable DOs. The build tool creates a configuration record for the DO, but does not write information about the DO into the VOB. Because scanning the information in the VOB is the only method other builds use to find DOs, other builds cannot wink in nonshareable DOs. However, a nonshareable DO can be reused by the view in which it was built.

A nonshareable DO can have shareable sub-DOs, but not shareable siblings. A nonshareable DO can be built using a winked-in shareable DO. (However, a shareable DO cannot have nonshareable sub-DOs or siblings.)

For information about enabling express builds, see *Using Express Builds to Prevent Winkin to Other Views* on page 55.

You can use the same commands that you use with shareable DOs on nonshareable DOs, but some commands work differently on the two kinds of DOs. The reference pages for the commands describe the differences.

Storage of Derived Objects

When a DO is created, its data container is located in the view storage area. For a shareable DO, the ClearCase build tool creates the VOB database object for the DO; it also writes to the VOB information about the DO that can be used during configuration lookup. A nonshareable DO has no VOB database object, and the build tool does not write any configuration lookup information into the VOB (Figure 6).

A DO consists of the following parts:

- VOB database object (shareable DOs only). Each DO is cataloged in the VOB database, where it is identified by an extended name that includes both its standard pathname (for example, /vobs/hw/src/hello.c) and a unique DO ID (for example, `23-Feb.08:41.391`).
- Data container. The data portion of a derived object is stored in a standard file within a ClearCase storage area. This file is called a *data container*; it contains the DO's file system data.
- Configuration record. Actually, a CR is associated with a DO; it is not part of the DO itself. More precisely, a CR is associated with the entire set of sibling DOs created by a particular invocation of a particular build script. See *Configuration Records* on page 13.

When a shareable DO is first created, it is unshared:

- It appears only in that view.
- Its data container is a file in the view's private storage area.
- **clearmake** writes information about the DO into the VOB.

Promotion and Winkin

The first time a shareable derived object is winked in by another dynamic view or when either kind of DO is promoted manually with a **winkin** or **view_scrubber –p** command, its status changes to shared:

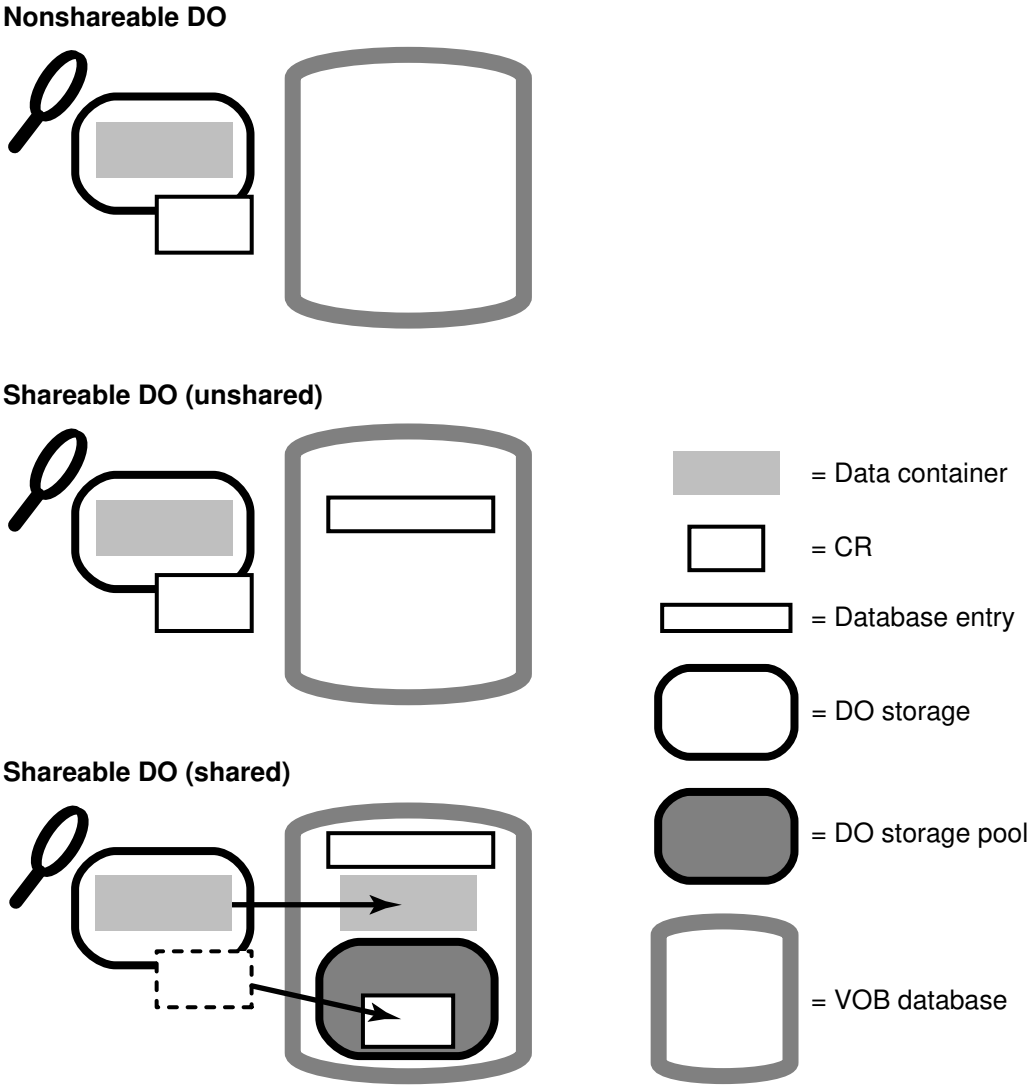
- Its data container is *promoted* to a derived object storage pool in the VOB.
- (Shareable DOs only) If the winkin was done by the build tool or the command was executed in another view, the DO now appears in two dynamic views.

When the winkin occurs during a **clearmake** build:

- The dynamic view to which the DO is winked in, and all other views to which the DO is subsequently winked in, use the data container in VOB storage.
- The original view continues to use the data container in view storage. (The **view_scrubber** utility removes this asymmetry, which causes all dynamic views to use the data container in VOB storage.)

When the winkin is done with the **winkin** or **view_scrubber –p** command, the data container in the view is removed after it is promoted to VOB storage. The original view and all other views to which the DO is subsequently winked in use the data container in VOB storage.

Figure 6 Storage of a Shareable Derived Object



After a derived object is winked in, it remains shared, no matter how many times it is winked in to additional dynamic views, and even if subsequent rebuilds or deletion commands cause it to appear in only one dynamic view (or zero views).

When a derived object's data container is in the VOB, any number of views can share the derived object without having to communicate with each other directly. For example, view **alpha** can be unaware of views **beta** and **gamma**, with which it shares

a derived object. The hosts on which the view storage directories are located need not have network access to each other's disk storage.

If **clearmake** attempts a winkin that fails, it checks to see if any VOBs are locked. If it finds a locked VOB, it waits until the VOB is unlocked and then retries the winkin.

For more information, see the **winkin** and **view_scrubber** reference pages.

DO Versions

You can check in a derived object as a version of an element, creating a *DO version*. Other versions of such an element can also be, but need not be, derived objects. A DO version behaves like both a version and a derived object:

- You can use its version ID to reference it as both a VOB database object and a data file.
- You can apply a version label to it and use that label to reference it.
- You can display its configuration record with **catcr** or compare the CR to another with **diffcr**.
- A **clearmake** build can wink it in if the element is located at the same pathname where the DO was originally built.
- You can wink it in with a **winkin** command.
- The **describe** command lists it as a `derived object version`. (The **lsdo** command does not list it at all.)

For more information about DO versions, see *Working with DO Versions* on page 57.

Reuse of DO IDs

The DO ID for a shareable derived object is guaranteed to be unique within the VOB, for all views. That is, if you delete a shareable DO, its numeric file name extension is not reused (unless you reformat the VOB that contains it).

The DO ID for a DO created by an express build (a nonshareable derived object) is unique only at a certain point in time. If you delete a nonshareable DO, the ClearCase build tools can reuse its numeric file name extension. (Because ClearCase tracks derived objects using their VOB database identifiers, no build confusion occurs if a file name extension is reused.)

DO IDs change when any of these events occur:

- The DO passes its first birthday. The time stamp changes to include the year the DO was created:

  `util.o@@15-Jul.15:34.8896` *(when first created)*

  `util.o@@15-Jul-2002.8896` *(after a year)*

- You convert a nonshareable DO to a shareable DO. (See *Converting Nonshareable DOs to Shared DOs* on page 62.)

- You process a VOB's database with **reformatvob**. All DO-IDs receive new numeric file-name extensions:

  `util.o@@15-Jul.15:34.8896` *(before reformatvob)*

  `util.o@@15-Jul.17:08.734` *(after reformatvob)*

The configuration record reflects these DO-ID changes.

Derived Object Reference Counts

A DO's reference count is the number of times the derived object appears in ClearCase dynamic views throughout the network. ClearCase also tracks the identifiers for the views that reference the DO. When a new derived object is created, **clearmake** sets its reference count to **1**, indicating that it is visible in one view. Thereafter, each winkin of the DO to an additional view increments the reference count.

The **lsdo –long** command lists the reference count and referencing views for a DO. For example:

cleartool lsdo –long

```
01-Sep-03.18:56:45     Suzanne Lee (sgl.user@neon)
  create derived object "file.txt@@01-Sep.18:56.2147483683"
  size of derived object is: 10
  last access: 01-Sep-03.18:56:46
  references: 1  => neon:/home/sgl/views/sgl_test.vws
01-Sep-03.19:03:19     Suzanne Lee (sgl.user@neon)
  create derived object "util@@01-Sep.19:03.81"
  size of derived object is: 10
  last access: 01-Sep-03.19:03:33
  references: 2 (shared)
  => neon:/home/sgl/views/sgl_test.vws
  => neon:/home/sgl/views/point_of.vws
```

For a nonshareable DO, the reference count is always **1**.

You can also create OS-level hard links to an existing shareable DO, each of which increments the reference count. Such additional hard links are sometimes subject to winkin:

- If the additional hard link was created in the same build script as the original DO, a winkin of the DO during a subsequent **clearmake** build causes a winkin of the additional hard link.
- Additional hard links that you create manually are not winked in during subsequent builds.

Note: Symbolic links are not subject to winkin, and **clearmake** regards symbolic links that point to the same object as being identical, whether or not the symbolic links are VOB links or view-private links.

A reference count can also decrease. When a program running in any of the views that reference a shared derived object overwrites or deletes that object, the link is broken and the reference count is decremented. That is, the program deletes the view's reference to the DO, but the DO itself remains in VOB storage. This occurs most often when a compiler overwrites an old build target. You can also remove the derived object with a standard **rm** command, or if the makefile has a **clean** rule, by running **clearmake clean**.

A derived object's reference count can become zero. For example, suppose you build program hello and rebuild it a few minutes later. The second hello overwrites the first hello, decrementing its reference count. Because the reference count probably was **1** (no other view has winked it in), it now becomes **0**. Similarly, the reference counts of old DOs, even of DOs that are widely shared, eventually decrease to zero as development proceeds and new DOs replace the old ones.

The **lsdo** command ignores such DOs by default, but you can use the **–zero** option to list them:

cleartool lsdo –zero –long hello.o

```
  .
  .
08-Mar-03.12:47:54     Allison K. Pak (akp.user@cobalt)
  create derived object "hello.o@@08-Mar.12:47.259"
  references: 0
  ...
```

A derived object that is listed with a `references: 0` annotation does not currently appear in any view. However, some or all of its information may still be available:

- If the DO was ever promoted to VOB storage, its data container is still in the VOB storage pool (unless it has been scrubbed), and its CR is still in the VOB database. You can use **catcr** and **diffcr** to work with the CR. You can get to its file system data by performing a **clearmake** build in an appropriately configured view or by using the **winkin** command.
- If the DO was never promoted, its CR may be gone forever. Until the scrubber runs and deletes the data container, the **catcr** command prints the message `Config record data no longer available for DO-pname`.

Pointers on Using ClearCase Build Tools

3

This chapter presents some pointers on making best use of **clearmake**.

Invoking clearmake

You can invoke **clearmake** from the command line or from the ClearCase File Browser (**xclearcase**). The command-line interface is designed to be as similar as possible to other **make** variants. Lowercase, single-letter command options have their familiar meanings. For example:

–n	No-execute mode
–f	Specify name of makefile
–u	Unconditional rebuild

clearmake recognizes additional options (also one letter, but uppercase) that control its enhanced functionality: configuration lookup, creation of configuration records and derived objects, parallel building, and so on. For a complete description, see the **clearmake** reference page.

You can run **clearmake** as a background process or invoke it from a shell script. (In **clearmake** output, some names are in bold, for clarity. On some architectures, running **clearmake** in the background suppresses the bold, but no characters are lost.)

A Simple clearmake Build Scenario

clearmake is designed to let developers in makefile-based build environments continue working in their accustomed manner. The following simple build scenario demonstrates how little adjustment is required to begin building with **clearmake**.

1 Set a view. Because working with ClearCase data requires a view context, it makes sense to set a view before starting a build.

(Strictly speaking, this is not required: if your process has a working directory view context, but not a set view context, **clearmake** sets the view to the working

directory view by executing a **cleartool setview –exec clearmake** command. If your process has a working directory view context and a set view context, **clearmake** uses the working directory view.

2. Go to a development directory within any VOB.
3. Edit some source files. Typically, you need to edit some sources before performing a build; accordingly, you check out some file elements and revise the checked-out versions.
4. Start a build. You can use your existing makefiles, but invoke **clearmake** instead of your standard **make** program. For example:

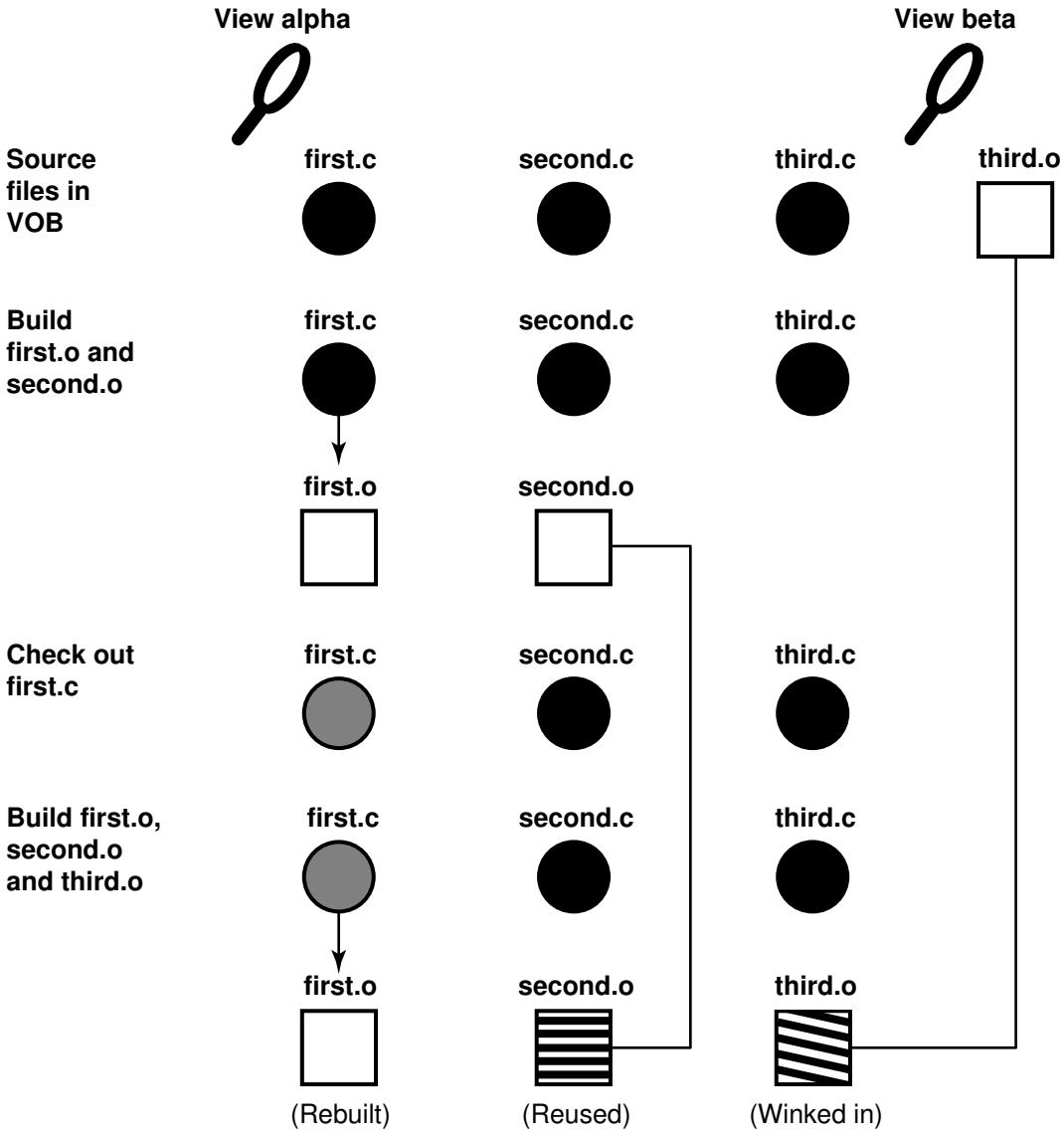
clearmake	*(build the default target)*
clearmake cwd.o libproj.lib	*(build one or more particular targets)*
clearmake –k monet CFLAGS=–g	*(use standard options and make-macro overrides)*

   (We recommend that you avoid specifying **make**-macro overrides on the command line. See *Specifying Build Options* on page 31.)

clearmake builds targets (or avoids building them) in a manner similar to, but more sophisticated than, other **make** variants.

Figure 7 illustrates some typical build scenarios in which derived objects are rebuilt, reused, or winked in.

Figure 7 clearmake Build Scenario



clearmake builds new derived objects for checked-out source files, reuses derived objects for checked-in source files that have previously built the object, and winks in derived objects from other views as appropriate for checked-in source files that have not previously built the object.

Note that **clearmake** does not attempt to verify that you have actually changed the file; the checkout makes a rebuild necessary. As you work, saving a file or invoking **clearmake** causes a rebuild of the updated file's dependents, in the standard **make** manner.

For source files that you have not checked out, **clearmake** may or may not build a new derived object:

- It may reuse a derived object that appears in your view, produced by a previous build.
- It may wink in an existing derived object built in another view. (It's even possible that a winked-in DO was originally created in your view, shared, and then deleted from your view, for example, by a **make clean** command.)
- Changes to other aspects of your build environment may trigger a **clearmake** rebuild: revision to a header file; change to the build script, use of a **make**-macro override; change to an environment variable used in the build script.

Accommodating Build Avoidance

When you first begin to build software systems with Rational ClearCase, the fact that **clearmake** uses a different build-avoidance algorithm than other **make** variants may occasionally surprise you. This section describes several such situations and presents simple techniques for handling them.

Increasing the Verbosity Level of a Build

If you do not understand **clearmake**'s build-avoidance decisions, use the **–v** (somewhat verbose) or **–d** (extremely verbose) option or set environment variable CCASE__VERBOSITY to **1** or **2**, respectively. When **clearmake** rebuilds a target because its build scripts do not match, the **-v** and **-d** options also return a summary of differences between the build scripts.

Handling Temporary Changes in the Build Procedure

Typically, you do not edit a target's build script in the makefile very often. But you may often change the build script by specifying overrides for **make** macros, either on the command line or in the UNIX environment. For example, target hello.o is specified as follows in the makefile:

```
hello.o: hello.c hello.h
       rm hello.o
       cc –c $(CFLAGS) hello.c
```

When it executes this build script, **clearmake** enters the build script, after macro substitution, into the config record. The command

% clearmake hello.o CFLAGS="–g –O1"

produces this configuration record entry:

```
--------------------------------
Build script:
--------------------------------
        cc -c -g -O1 hello.c
```

So does this command:

```
env CFLAGS="-g -O1" clearmake -e hello
```

The **clearmake** build-avoidance algorithm compares effective build scripts. If you then use the command **clearmake hello.o** without specifying **CFLAGS="–g –O1"**, **clearmake** rejects the existing derived object, which was built with those flags. The same mismatch occurs if you create a CFLAGS environment variable with a different value, and then invoke **clearmake** with the **–e** option.

Specifying Build Options

To manage temporary overrides for **make** macros and environment variables, place macro definitions in build options specification (BOS) files. **clearmake** provides several ways for using a BOS file. For example, if your makefile is named project.mk, macro definitions are read from project.mk.options. You can also keep a BOS file in your home directory or specify one or more BOS files with **clearmake –A**. For details, see *Build Options Specification Files* on page 70.

Using a BOS file to specify **make** macro overrides preserves these options, which makes it easier to reuse them. If you do not modify the BOS file frequently, derived objects in your view are not disqualified for reuse on the basis of build script discrepancies. Some of the sections that follow describe other applications of BOS files.

Handling Targets Built in Multiple Ways

Because **clearmake** compares build scripts, undesirable results may occur if your build environment includes more than one way to build a particular target. For example, suppose that the target test_prog_3 appears in two makefiles in two directories. The first is in its source directory, util_src:

```
test_prog_3: ...
       cc -o test_prog_3 ...
```

The second is in another directory, app_src:

```
../util_src/test_prog_3: ...
       cd ../util_src ; cc -o  test_prog_3
```

Derived objects built with these scripts may be equivalent, because they are built as the same file name (test_prog_3) in the same VOB directory (util_src). But by default, a build in the app_src directory never reuses or winks in a DO built in the util_src directory, because build-script comparison fails.

You can suppress build-script comparison for this target by using a **clearmake** special build target, **.NO_CMP_SCRIPT** in the makefile or in an associated BOS file:

```
.NO_CMP_SCRIPT: ../util_src/test_prog_3
```

To suspend build-script comparison once, you can use **clearmake –O**.

Using a Recursive Invocation of clearmake

You can eliminate the problem of different build scripts described in *Handling Targets Built in Multiple Ways* by adding a recursive invocation of **clearmake** to the makefile in app_src:

```
../util_src/test_prog_3: ...

     cd ../util_src ; $(MAKE) test_prog_3        ($(MAKE) invokes clearmake
                                                  recursively)
```

Now, target test_prog_3 is built the same way in both directories. You can turn on build-script comparison again, by removing the **.NO_CMP_SCRIPT** special target.

Optimizing Winkin by Avoiding Pseudotargets

Like other **make** variants, **clearmake** always executes the build script for a pseudotarget, a target that does not name a file system object built by the script. For example, in the section *Using a Recursive Invocation of clearmake*, you may be tempted to use a pseudotarget in the app_src directory's makefile:

```
test_prog_3: ...                          (shortened from ../util_src/test_prog_3)
      cd ../util_src ; $(MAKE) test_prog_3
```

A build of any higher-level target that has test_prog_3 as a build dependency always builds a new test_prog_3, which in turn triggers a rebuild of the higher-level target. If the rebuild of test_prog_3 was not necessary, the rebuild of the higher-level target may not have been necessary, either. Such unnecessary rebuilds decrease the extent to which you can take advantage of derived object sharing.

Accommodating the Build Tool's Different Name

The fact that the ClearCase build utility has a unique name, **clearmake**, may conflict with existing build procedures that implement recursive builds. Most make variants

automatically define the **make** macro **$(MAKE)** as the name of the build program, as it was typed on the command line:

% **make hello.o**	*(sets MAKE to "make")*
% **clearmake hello.o**	*(sets MAKE to "clearmake")*
% **my_make hello.o**	*(sets MAKE to "my_make")*

This definition enables recursive builds to use **$(MAKE)** to invoke the same build program at each level. The section *Optimizing Winkin by Avoiding Pseudotargets* includes one such example; here is another one:

```
SUBDIRS = lib util src

all:
       for DIR in $(SUBDIRS) ; do ( cd $$DIR ; $(MAKE) all ) ; done
```

Executing this build script with **clearmake** invokes **clearmake all** recursively in each subdirectory.

Declaring Source Dependencies in Makefiles

To implement build avoidance based on time stamps, standard **make** variants require you to declare all the source file dependencies of each build target. For example, object module hello.o depends on source files hello.c and hello.h in the same directory:

```
hello.o: hello.c hello.h
      rm -f hello.o
      cc -ccl /c hello.c
```

Typically, these source files depend on project-specific header files through **`#include`** directives, perhaps nested within one another. The standard UNIX files do not change very often, but programmers often lament that "it didn't compile because someone changed the project's header files without telling me."

To alleviate this problem, some organizations include every header file dependency in their makefiles. They rely on utility programs (for example, **makedepend**) to read the source files and determine the dependencies.

clearmake does not require that source-file dependencies be declared in makefiles (but see *Source Dependencies Declared Explicitly* on page 34). The first time a derived object is built, its build script is always executed; thus, the dependency declarations are irrelevant for determining whether the target is out of date. After a derived object has been built, its configuration record provides a complete list of source-file dependencies used in the previous build, including those on all header files (nested and nonnested) read during the build.

Note: clearmake will rebuild a target if there have been changes to any directories listed as source-file dependencies.

You can leave source-file dependency declarations in your existing makefiles, but you need not update them as you revise the makefiles. And you need not place source-file dependencies in new makefiles to be used with **clearmake**.

Note: Although source-file dependency declarations are not required, you may want to include them in your makefiles, anyway. The principal reason for doing so is portability: you may need to provide your sources to another team (or another company) that is not using ClearCase.

Source Dependencies Declared Explicitly

The ClearCase build auditing facility tracks only the MVFS objects used to build a target. Sometimes, however, you may want to track other objects. For example:

- The version of a compiler that is not stored in a VOB
- The version of the operating system kernel, which is not referenced at all during the build
- The state of a flag-file, used to force rebuilds

You can force such objects to be recorded in the CR by declaring them as dependencies of the makefile target:

```
hello.o: hello.c hello.h /usr/5bin/cc my.flag
      rm -f hello.o
      cc -c hello.c
```

This example illustrates dependency declarations for these kinds of objects:

- (hello.c, hello.h) - Dependencies on MVFS objects are optional. These are recorded by clearmake and MVFS anyway.
- /usr/5bin/cc - Dependencies on build tools are required to track the build tools that are not stored in VOBs. These dependencies are listed as non-MVFS objects in the configuration record.
- my.flag - Dependencies on view-private objects can implement a flag-file capability.

  We suggest that you use view-private files as flag files, rather than using non-MVFS files (such as /tmp/flag). In a parallel build, a view-private flag file is guaranteed to be the same object on all hosts; there is no such guarantee for a non-MVFS file.

As an alternative to declaring your C compiler as a build dependency, you can place it (and other tools) in a tools VOB. The versions of such tools are recorded, eliminating the need for explicit dependency declarations. Additional issues in the auditing of build tools are discussed in the section *Explicit Dependencies on Searched-For Sources*.

Explicit Dependencies on Searched-For Sources

There are situations in which the configuration lookup algorithm that **clearmake** uses qualifies a derived object, even though rebuilding the target would produce a different result. Configuration lookup requires that for each object listed in an existing CR, the current view must select the same version of that object. However, when search paths must be used to find an object, a target rebuild may use a different object than the one listed in the CR. Configuration lookup does not take this possibility into account.

When files are accessed by explicit pathnames, configuration lookup qualifies derived objects correctly. Configuration lookup may qualify a derived object incorrectly if files are accessed at build time by a search through multiple directories, for example, when the **–I** option to a C or C++ compiler specifies a header file or when the **–L** option to a linker specifies a library file. The following build script uses a search to locate a library file, libprojutil.a:

```
hello:
      cc –o hello –L /usr/project/lib –L /usr/local/lib \
            main.o util.o –lprojutil
```

The command **clearmake hello** may qualify an existing derived object built with /usr/local/lib/libprojutil.a, even though rebuilding the target would now use /usr/project/lib/libprojutil.a instead.

clearmake addresses this problem in the same way as some standard **make** implementations:

- You must declare the searched-for source object as an explicit dependency in the makefile:

  ```
  hello: libprojutil.a
  ...
  ```

- You must use the **VPATH** macro to specify the set of directories to be searched:

  ```
  VPATH = /usr/project/lib:/usr/local/lib
  ```

Given this makefile, **clearmake** uses the **VPATH** (if any) when it performs configuration lookup on libprojutil.a. If a candidate derived object was built with /usr/local/lib/projutil.a, but would be built with /usr/project/lib/projutil.a in the current view, the candidate is rejected.

Note: The **VPATH** macro is not used for all source dependencies listed in the config record. It is used only for explicitly declared dependencies of the target. Also, **clearmake** searches only in the current view.

Build Tool Dependencies. You can use this mechanism to implement dependencies on build tools. For example, you can track the version of the C compiler used in a build as follows:

```
msg.o: msg.c $(CC)
       $(CC) -c msg.c
```

With this makefile, either your **VPATH** must include the directories on your search path (if the **$(CC)** value is **cc**), or you must use a full pathname as the **$(CC)** value.

Note: If your C compiler is stored in a VOB and you invoke it from the VOB, ClearCase tracks its version and you do not have to include it as a dependency.

Build-Order Dependencies

In addition to source dependencies, makefiles also contain build-order dependencies. For example:

```
hello: hello.o libhello.a
      ...
libhello.a: hello_env.o hello_time.o
      ...
```

These dependencies are buildable objects known as *subtargets*. The executable hello must be built after its subtargets, object module hello.o and library libhello.a, and the library must be built after its subtargets, object modules hello_env.o and hello_time.o.

ClearCase does not detect build-order dependencies; you must include such dependencies in makefiles used with **clearmake**, as you do with other **make** variants.

Problems with Forced Builds

clearmake has a **–u** option (unconditional), which forces rebuilds. Using this option reduces the efficiency of derived object sharing, however. If you force **clearmake** to build a target when it would have winked in an existing DO, you create a new DO with the same configuration as an existing one. In such situations, a developer who expects a build to share a particular existing DO may get another, identically configured DO instead. This may confuse the team and waste disk space.

We suggest that you use a flag file to force a rebuild, rather than using **clearmake –u**. (See *Source Dependencies Declared Explicitly* on page 34.)

How clearmake Interprets Double-Colon Rules

Double-colon rules are a special kind of makefile construct that allows several independent rules for one target, each with a possibly different build script. The semantics given to these rules by other **make** programs (such as Gnu make, Sun make,

and for **clearmake** when CRs are not being generated) are that commands within each double-colon rule are executed if the target is older than any dependencies of that particular rule. The result can be that none, any, or all of the double-colon rules are executed.

However, when **clearmake** creates CRs and associates them with the results of its builds, this interpretation runs the risk of generating incomplete CRs, which do not contain all the versions and build scripts used to build the targets. For this reason, **clearmake** interprets these rules in a more conservative way.

When building a target specified by a number of double-colon rules, **clearmake** concatenates all build scripts from all the double-colon rules for that target and runs them in a single audited script.

To produce the correct results, any subtargets must already have been built, so **clearmake** builds any out-of-date subtargets before it executes the concatenated build script.

As a result, you may observe these differences in behavior between **clearmake** and other **make** programs concerning double-colon rules:

- **clearmake** runs more of the build scripts than other **make** programs.
- **clearmake** may run the build scripts in a different order than other **make** programs.

However, given the intended use and standard interpretation of double-colon rules, these differences still produce correct builds and complete correct CRs.

Continuing to Work During a Build

As your build progresses, other developers continue to work on their files and may check in new versions of elements that your build uses. If your build takes an hour to complete, you do not want build scripts executed early in the build to use version 6 of a header file and scripts executed later to use version 7 or 8.

To prevent such inconsistencies, any version whose selection is based on a **LATEST** config spec rule is locked out if it is checked in after the instant that **clearmake** was invoked. The moment that the **clearmake** build session begins is the build reference time.

The same reference time is reported in each configuration record produced during the build session, even if the session lasts hours (or days):

```
% cleartool catcr hello.o
Target hello.o built by drp.dvt
Host "fermi" running OSF1 V1.3 (alpha)
Reference Time 26-Feb-03.16:53:58, this audit started
26-Feb-03.16:54:10 ...
```

Note: The `reference time` is the build reference time, when the overall **clearmake** build session began. The `this audit started` time is when the execution of the individual build script began.

When determining whether an object was created before or after the build reference time, **clearmake** adjusts for clock skew, the inevitable small differences among the system clocks on different hosts. For more information about build sessions, see *Build Sessions, Subsessions, and Hierarchical Builds* on page 39.

Caution: A build's coordinated reference time applies to elements only, providing protection from changes made after the build began. You are not protected from changes to view-private objects and non-MVFS objects. For example, if you begin a build and then change a checked-out file used in the build, a failure may result. To avoid this problem, do not work on the same project in a view in which a build is in progress.

Using Config Spec Time Rules

Note: If you use a UCM view, your config spec is generated by ClearCase. Do not add time rules to your config spec.

Using the reference time facility described in *Continuing to Work During a Build*, **clearmake** blocks out potentially incompatible source-level changes that take place after your build begins. For example, if a rule contains a remote branch type and a synchronization takes place during a build, **clearmake** uses the latest version of the remote files that were checked in before the time lock. If, however, an incompatible change has already taken place, ClearCase allows you to block out recently created versions.

A typical ClearCase development strategy is for each team member to work in a separate view, but to have all the views use the same config spec. In this way, the entire team works on the same branch. As long as a source file remains checked out, its changes are isolated to a single view; when a developer checks in a new version, the entire team sees it on the dedicated branch.

This incremental integration strategy is often very effective. But suppose that another user's recently checked-in version causes your builds to start failing. Through an exchange of e-mail, you trace the problem to header file project_base.h, checked in at 11:18 A.M. today. You and other team members can reconfigure your views to roll back that one element to a safe version:

```
element project_base.h .../onyx_port/LATEST -time 5-Mar.11:00
```

If many interdependent files have been revised, you can roll back the view for all checked-in elements:

```
element * .../onyx_port/LATEST -time 5-Mar.11:00
```

For a complete description of time rules, see the **config_spec** reference page.

Inappropriate Use of Time Rules

Your view interprets time rules with respect to the `create version` event record written by the **checkin** command. The checkin is read from the system clock on the VOB server host. If that clock is out of sync with the clock on the view server host, your attempt to roll back the clock may fail. Thus, do not strive for extreme precision with time rules: select a time that is well before the actual cutoff time (for example, a full hour before, or in the middle of the night).

Do not use time rules to freeze a view to the current time immediately before you start a build. Allow **clearmake**'s reference time facility to perform this service. Here's an inappropriate use scenario:

1. You check in version 12 of util.c at 7:05 P.M. on your host. You do not know that clock skew on the VOB host causes the time 7:23 P.M. to be entered in the `create version` event record.
2. To freeze your view, you change your config spec to include this rule:

   ```
   element * /main/LATEST -time 19:05
   ```

3. You issue a **clearmake** command immediately (at 7:06 P.M.) to build a program that uses util.c. When selecting a version of this element to use in the build, your view consults the event history of util.c and rejects version 12, because the 7:23 P.M. time stamp is too late for the **–time** configuration rule.

Build Sessions, Subsessions, and Hierarchical Builds

The following terms are used to describe the details of ClearCase build auditing:

- Invoking **clearmake** or **clearaudit** starts a build session. The time at which the build session begins becomes the build reference time for the entire build session, as described on *Continuing to Work During a Build* on page 37.
- During a build session, one or more target rebuilds typically take place.
- Each target rebuild involves the execution of one or more build scripts. (A double-colon target can have multiple build scripts; see *How clearmake Interprets Double-Colon Rules* on page 36.)
- During each target rebuild, **clearmake** or **clearaudit** conducts a build audit.

Subsessions

A build session can have any number of subsessions, all of which inherit the reference time of the build session. A subsession corresponds to a nested build or recursive make, which is started when a **clearmake** or **clearaudit** process is invoked in the process family of a higher-level **clearmake** or **clearaudit**. For example:

- Including a **clearmake** or **clearaudit** command in a makefile build script executed by **clearmake** or **clearaudit**
- Entering a **clearmake** or **clearaudit** command in an interactive process started by **clearaudit**

A subsession begins while a higher-level session is still conducting build audits. The subsession conducts its own build audits, independent of the audits of the higher-level session; that is, the audits are not nested or related in any way, other than that they share the same build reference time.

Versions Created During a Build Session

Any version created during a build session and selected by a **LATEST** config spec rule is not visible in that build session. For example, a build checks in a derived object that it has created; subsequent commands in the same build session do not select the checked-in version, unless it is selected by a config spec rule that does not use the version label **LATEST**.

An effect of this behavior is that you cannot check in and label a version during a single build session. Instead, you must check in the version during one build session, and label it during another build session. Use the **mklabel –config** command to label versions associated with a specific derived object.

Coordinating Reference Times of Several Builds

Different build sessions have different reference times. The best way to assign a series of builds the same reference time is to structure them as a single, hierarchical build.

An alternative approach is to run all the builds within the same **clearaudit** session. For example, you can write a shell script, **multi_make**, that includes several invocations of **clearmake** or **clearaudit** (along with other commands). Running the script as follows ensures that all the builds are subsessions that share the same reference time:

clearaudit –c multi_make

Objects Written at More Than One Level

Problems occur when the same file is written at two or more session levels (for example, a top-level build session and a subsession): the build audit for the

higher-level session does not contain complete information about the file system operations that affected the file. For example:

clearaudit –c "clearmake shuffle > logfile"

The file logfile may be written twice:

- During the **clearaudit** build session, by the shell program invoked from **clearaudit**
- During the **clearmake** subsession, when the **clearaudit** build session is suspended

In this case, **clearaudit** issues this error message:

```
clearaudit: Error: Derived object modified; cannot be stored in VOB.
Interference from another process?
```

To work around this limitation, postprocess the derived object at the higher level with a **cp** command:

clearaudit –c "clearmake shuffle > log.tmp; cp log.tmp logfile; rm log.tmp"

Build Auditing and Background Processes

The ClearCase build programs—**clearmake**, **clearaudit**, and **abe**—use the same procedure to produce configuration records:

1. Send a request to the host's multiversion file system (MVFS), to initiate build auditing.
2. Start one or more child processes (typically, shell processes), in which makefile build scripts or other commands are executed.
3. Turn off MVFS build auditing.
4. If all the subprocesses have indicated success and at least one MVFS file has been created, compute and store one or more configuration records.

Any subprocesses of the child processes started in Step 2 inherit the same MVFS build audit. (Recursive invocations of ClearCase build programs conduct their own, independent audits; see *Build Sessions, Subsessions, and Hierarchical Builds* on page 39.)

A problem can occur if a build script (or other audited command) invokes a background subprocess and exits without waiting for it to complete. The build program has no knowledge of the background process and may proceed to Step 3 and Step 4 before the background process has finished its work. In such situations, ClearCase cannot guarantee what portion, if any, of the actions of background commands will be included in the resulting CR. The contents of the CR depend on system scheduling and timing behavior.

The ClearCase build programs audit background processes correctly only if both of the following conditions are true:

- The build script does not complete until all background processes are known to have finished.
- Each background process performs its first MVFS file access while it is still a descendant process of the **clearmake** or **clearaudit** process. (The ClearCase kernel component determines whether to audit a given process when that process first accesses the MVFS. If the process's ancestors include a process already being audited, the descendant process is similarly marked for auditing.)

If either or both of these conditions are false, avoid using background processes in audited build scripts.

Working with Incremental Update Tools

The design of the build auditing capability makes it ideal for use with tools that build derived objects from scratch. Because newly created objects have no history, ClearCase can learn everything it needs to know at build time. But this reliance on auditing at the files system level at build time can cause ClearCase to record incomplete information for objects that are updated incrementally, which do have a history.

In ClearCase, incremental updating means that an object is updated partially during the builds of multiple makefile targets, instead of generated completely by the build of one target. By default, **clearmake** does not update an existing CR incrementally when it builds a target. Instead, it does the following:

- Each time a build script incrementally updates an object's file system data, **clearmake** writes a completely new CR, which describes only the most recent update, not the entire build history.
- The new CR does not match the desired build configuration for any of the other targets that update the object incrementally.

The result is a situation that is both unstable and incorrect: all incremental-update targets are rebuilt each time that **clearmake** is invoked; when the build is finished, the DO has the correct file system data, but its CR may not describe the DO's configuration accurately.

clearmake provides a special makefile target **.INCREMENTAL_TARGET**, which can be used to guarantee correct CR information for incremental updates. The following sections give examples of how to use **.INCREMENTAL_TARGET**.

Example: Building an Archive

A common incremental-update scenario is the building of an archive by **ar(1)**. A traditional **make** program treats an archive as a compound object; it can examine the

time stamps of the individual components (object modules) in the archive; and it can update the archive by replacing one or more individual object modules. Here is a simple makefile in which a special syntax enables multiple targets to update a single archive, `libvg.a`, incrementally:

```
libvg.a:: libvg.a(base.o)
libvg.a:: libvg.a(in.o)
libvg.a:: libvg.a(out.o)
```

If you edit one of the library's sources (for example, **out.c**), a traditional **make** program uses the special syntax and a **.c.a** built-in rule to update the library as follows:

1. It looks inside the archive libvg.a, and determines that it includes an **out.o** that is older than its source file.
2. It compiles a new out.o from out.c.
3. It uses **ar** to incrementally update libvg.a, replacing the old instance of object module out.o with the newly built instance.

clearmake does not implement this algorithm and includes no support for treating an archive as a compound object. ClearCase build-avoidance is based solely on metadata (CRs), not on any analysis of the file system data. **clearmake** interprets the above makefile as follows:

1. It considers all the libvg.a(...) dependencies to be multiple instances of the same double-colon build target.
2. Accordingly, whenever one of those double-colon targets requires rebuilding, **clearmake** rebuilds them all, using the standard **.c.a** built-in rule. The effect is to rebuild the entire archive libvg.a from scratch.

Thus, **clearmake** accepts the standard incremental-update syntax, but interprets it in a way that produces a nonincremental build procedure.

Makefile Restructuring for Incremental Archive Targets

The makefile in the previous example can be restructured to allow incremental updates to an archive:

```
.INCREMENTAL_TARGET: libvg.a
libvg.a: base.o in.o out.o
       ar rv libvg.a $?

base.o:
       cc -c base.c

in.o:
       cc -c in.c

out.o:
       cc -c out.c
```

Object modules built by this makefile are standard, shareable derived objects; typically, as library sources stabilize, most builds of target libvg.a reuse or wink in most of the object modules.

The **.INCREMENTAL_TARGET** directive tells **clearmake** to merge CRs incrementally for this target, that is, to merge the dependencies listed in the previous CR with those of the latest build. In this way, no information is lost from the CRs. Note that **.INCREMENTAL_TARGET** accepts patterns on its list of targets, as well as file names, so you can direct **clearmake** to merge all archive targets incrementally by including the directive **.INCREMENTAL_TARGET: %.a** in the makefile.

During a rebuild, the **$?** macro expands to the list of dependencies that do not match the current configuration record for the target. **$?** is useful in conjunction with **ar r** to replace, in the archive library, only those objects that have changed.

Avoid the following alternate restructuring; it causes a complete rebuild of the archive each time any object module is updated:

```
base.o: base.c
       cc -c base.c
       ar rv libvg.a base.o
  .
  .  and so on
```

Note: When you use **.INCREMENTAL_TARGET** with an archive library, the full set of declared dependencies must be the same in all makefiles that update that library. Do not attempt to build up libraries incrementally from two different makefiles. For example:

```
../lib.a: base.o                    (makefile 1)

ar rv ../lib.a base.o


../lib.a: in.o out.o                (makefile 2)

ar rv ../lib.a in.o out.o
```

The rules are executed in multiple steps, and **clearmake** doesn't combine them to verify that the target is up to date with respect to all of its dependencies. Using the construction given above can result in missing required rebuilds (with either **make** or **clearmake**).

A Note on the Use of ar Keys

Do not use the **u** key with **ar**; it is not reliable within a ClearCase environment.

The **r** key may be used to direct **ar** to replace one or more object modules in an archive library without replacing the entire library. The **u** key directs **ar** to replace only those object modules with modification dates more recent than the archive library.

This behavior creates problems within ClearCase. When determining whether a .o file needs to be rearchived, **ar** looks only at whether its time stamp is older than that of the .a file. This check is not sufficient to determine whether a file inside a ClearCase VOB is out of date. For example, a build winks in a .o file whose time stamp is older than the time stamp of the .a file. Because the file is different from the one used the last time you built the archive, you want the file to be rearchived. However, because **ar** sees that the time stamp is older, it does not rearchive the .o file.

Example: Incremental Linking

If your makefile is structured properly, configuration records are not likely to lose information during incremental links.

Incremental linkers typically work by determining which object files have changed since the last link and relinking only those objects. Because the linker may not read every object each time it links, a CR can, in theory, lose information as repeated links are made. But in practice, because all dependencies of the link are listed in the build script, the build script does not change from one link invocation to the next. And, because you typically list the objects or predefined dependencies of the link, those dependencies are included in the CR.

Additional Incremental-Update Situations

You may encounter incremental updating in other situations, as well. For example, C++ compilers that support parameterized types (templates) often update type map files incrementally as different targets are built. ClearCase includes special makefile rules that store per-target type map files. For more information, see Chapter 7, *Using ClearCase to Build C++ Programs*.

Ada compilers often update certain common files in Ada libraries incrementally, as different compilation units are built. There are no current **clearmake** workarounds to

implement per-target CRs for Ada libraries. To produce a CR for an Ada library, you can rebuild the library from its sources in a single **clearaudit** session.

Adding a Version String or Time Stamp to an Executable

This section describes simple techniques for incorporating a version string and/or time stamp into a C-language compiled executable. Including a version string or time stamp allows anyone (for example, a customer) to determine the exact version of a program by entering a shell command.

The techniques described below support use of the **what** command or the **–Ver** option. For example:

```
% what monet
monet R2.0 Baselevel 1
Thu Feb 11 17:33:23 EST 2003
% monet –Ver
monet R2.0 Baselevel 1 (Thu Feb 11 17:33:23 EST 2003)
```

After the particular version of the program is determined, you can use ClearCase commands to find a local copy, examine its config record, and if appropriate, reconstruct the source configuration with which it was built. (Presumably, the local copy is a derived object that has been checked in as a version of an element.)

You can identify the appropriate derived object by attaching a ClearCase attribute with the version string to the checked-in executable, or you can rely on the time stamp and your ability to run the **what** command on the checked-in executable to find it.

Creating a what String

The **what** program searches for a null-terminated string that starts with a special four-character sequence:

```
@(#)
```

To include this string in a C-language executable, define a global character-string variable. For example, these source statements produce the two-line **what** listing above:

```
char *version_string = "@(#)monet R2.0 Baselevel 1";
char *version_time   = "@(#)Thu Feb 11 17:33:23 EST 2003;
```

As an alternative, you can generate the time stamp dynamically when the **monet** program is linked:

1. Create a new source file that contains the statements that define the **what** strings. Instead of hard-coding a date string, use a **cpp(1)** macro in the source file. This

example uses a source file named version_info.h, which contains a macro named **DATE**:

```
char *version_string = "@(#)monet R2.0 Baselevel 1";
char *version_time = DATE;
```

2 Use shell command substitution to incorporate the current time dynamically into the value for the **DATE** macro:

```
SHELL = /bin/sh
OTHER_OBJS = main.o cmd_line.o (and so on)

monet: version_info.h $(OTHER_OBJS)
cc -o monet -DDATE="\"@(#)`date`\"" $(OTHER_OBJS)
```

A rebuild of **monet** is also triggered if the *version_string* variable is edited manually in **version_info.h**.

The *version_string* can be generated dynamically, too (for example, with environment variables). But it is more likely that the project manager edits this string's value before major builds.

Note: If you use **clearmake** to build **monet**, you need not declare version_info.h as an explicit dependency.

Implementing a –Ver Option

You need not depend on the **what** command to extract version information from your executable. Instead, you can have the program itself output the information stored in the *version_string* and *version_time* variables. Revise the source module that does command-line processing to support a what version option (for example, –**Ver**):

```
#include <stdio.h>

main(argc,argv)
    int argc;
    char **argv;
{
/*
 * implement -Ver option
 */
    if (argc > 1 && strcmp(argv[1],"-Ver") == 0) {
        extern char *version_string;
        extern char *version_time;
        /*
         * Print version info, skipping the "@(#)" characters
         */
        printf ("%s (%s)\n",
                    &version_string[4], &version_time[4]);
        exit(0);
    }
}
```

Working with DOs and Configuration Records

4

This chapter describes the operations you can perform on derived objects and configuration records. For more information and examples, see the reference pages for the commands.

The information in this chapter applies only to dynamic views.

Setting Correct Permissions for Derived Objects

If you and other members of your team want to share derived objects (DOs), make sure that your views are configured to create shareable DOs and that the DOs are created with a mode that grants both read and write access to team members. To accomplish this, use either of the following alternatives:

- Set your **umask** value to **2** in your shell startup file.
- Set the environment variable CCASE_BLD_UMASK to **2** and leave a more restrictive **umask** value in your **.login** file or its equivalent (the value **22** is commonly used, which denies write access to group members). When **clearmake** runs a build script, it performs the following steps:

  a Saves the current **umask** value.

  b Sets the **umask** value to the value of CCASE_BLD_UMASK.

  c Creates a shell (if necessary) to run the build script.

  d Restores the original **umask** value when the build script (or shell) completes.

  You can set CCASE_BLD_UMASK as a make macro, instead of as an environment variable.

  Note: If you want to use CCASE_BLD_UMASK, do not set your **umask** value in your shell startup file. If you set the **umask** value in your startup file, the **umask** value is reset to its original value in Step c when the startup file is read. Setting CCASE_BLD_UMASK in your startup file has no effect.

Other users cannot overwrite and destroy a DO that you are still using, even if you use a CCASE_BLD_UMASK value that grants write access to group members. If your DO has been winked in to another view, and the corresponding makefile target is rebuilt in that

view, **clearmake** first breaks the link to your DO, and then creates a file in that view for the build script to overwrite.

Permissions on DOs affect the extent to which they are shareable:

- When you perform a build, the ClearCase build tool winks in a derived object to your view only if you have read permission on the DO.
- The ClearCase build tool can wink in DOs for which you do not have write permission. But `permission denied` errors may occur during a subsequent build, when a compiler (or other build script command) attempts to overwrite such a DO. To work around this problem, you can rewrite your makefile to remove the target before rebuilding it. You can also set a policy for how users must set their permissions.

For information about fixing the permissions of DO versions, see the **protect** reference page.

Listing and Describing Derived Objects

The following sections describe how to use the **lsdo**, **describe**, **ls**, and **lsprivate** commands to list derived objects.

Listing Derived Objects Created at a Certain Pathname

Use the **lsdo** command to list derived objects created at a specific pathname. For information about the kinds of DOs included in the listing, see the **lsdo** reference page.

- To list all DOs created at the pathname adm.h:

  cleartool lsdo adm.h

  ```
  01-Jul.13:49 "adm.h@@01-Jul.13:49.1286781"
  30-Jun.20:03 "adm.h@@30-Jun.20:03.1278990"
  30-Jun.18:14 "adm.h@@30-Jun.18:14.1277470"
  29-Jun.19:11 "adm.h@@29-Jun.19:11.1253509"
  29-Jun.18:13 "adm.h@@29-Jun.18:13.1252790"
  29-Jun.16:09 "adm.h@@29-Jun.16:09.1249897"
  ```

- To list all DOs created by you at the pathname adm.h:

  cleartool lsdo –me adm.h

  ```
  30-Jun.18:14 "adm.h@@30-Jun.18:14.1277470"
  ```

Listing a Derived Object's Kind

To display a derived object's kind, use the **cleartool** commands **ls –l**, **lsprivate –l –do**, or **describe –fmt "%[DO_kind]p"**. The kind can be one of the following values:

nonshareable	The DO was created during an express build and cannot be winked in by other views.
unshared	The DO was created during a regular build. Its data container is located in view storage, not in the VOB.
promoted	The DO's data container has been promoted to the VOB by a **winkin** or **view_scrubber –p** command. The DO is referenced by only one view.
shared	The DO's data container has been promoted to the VOB by a ClearCase build tool or a manual **winkin** or **view_scrubber –p** command.

- List, in long form, a particular DO.

  cleartool ls –l util
  ```
  derived object (non-shareable)  util@@01-Sep.10:54.2147483681
  ```

- To list all DOs created in the current view in the **/vobs/dev** VOB, including the DO kind:

  cleartool lsprivate –long –invob /vobs/dev –do
  ```
  derived object (unshared)  /vobs/dev/file2.txt@@02-Jul.13:51.124
  derived object (unshared)  /vobs/dev/file2sub.txt@@02-Jul.13:51.123
  ```

- To list the name and kind of all DOs created in the current view:

  cleartool describe –fmt "%n\t%[DO_kind]p\n" `cleartool lsprivate –do`
  ```
  /vobs/dev/file2.txt@@02-Jul.13:51.124       shared
  /vobs/dev/file2sub.txt@@02-Jul.13:51.123    shared
  /vobs/dev/dir1/x.o@@01-Jul.14:23.186      unshared
  /vobs/dev/api/bin/adm.exe@@04-Jul.04:01.776 unshared
  ...
  ```

Displaying a DO's OID

A derived object's OID is the permanent identifier recorded in the VOB database for the DO. It does not change over the life of the DO, unlike the DO-ID (see *Reuse of DO IDs* on page 24). To display the OID, use the command **describe –fmt "%On"**. For example:

cleartool describe –fmt "%On\n" x.o
```
b7afc83e.2f2311d3.a382.00:01:80:7b:09:69
```

Displaying a Description of a DO Version

The **describe** command displays descriptions of DO versions, as it does descriptions of regular versions. You can use the **–fmt** option to extract parts of the description. For

example, the following command prints the name, predecessor version, and element type of a DO version:

cleartool describe –fmt "%n\t%[version_predecessor]p\t%[type]p\n" file1.o

```
file1.o@@/main/2     /main/1  text_file
```

For more information about the **–fmt** option, see the **fmt_ccase** reference page.

Identifying the Views That Reference a Derived Object

The VOB stores information about which views reference a derived object. To display this information, use the **lsdo** command:

cleartool lsdo –l hello.o

```
10-Mar-03.15:25:52    Allison K. Pak (akp.user@copper)
  create derived object "hello.o@@10-Mar.15:25.213"
  size of derived object is: 450
  last access: 15-Mar-03.14:22:17
  references: 2 (shared)
  => copper:/home/akp/tut/old.vws
  => copper:/home/akp/tut/fix.vws
```

Caching Unavailable Views

When **clearmake** shops for a derived object to wink in to a build, it may find DOs from a view that is unavailable (because the view server host is down, the **albd_server** is not running on the server host, and so on). Attempting to fetch the DO's configuration record from an unavailable view causes a long time-out, and the build may reference multiple DOs from the same view.

clearmake and other **cleartool** commands that access configuration records and DOs (**lsdo**, **describe**, **catcr**, **diffcr**) maintain a cache of tags of inaccessible views. For each view tag, the command records the time of the first unsuccessful contact. Before trying to access a view, the command checks the cache. If the view's tag is not listed in the cache, the command tries to contact the view. If the view's tag is listed in the cache, the command compares the time elapsed since the last attempt with the time-out period specified by the CCASE_DNVW_RETRY environment variable. If the elapsed time is greater than the time-out period, the command removes the view tag from the cache and tries to contact the view again.

Note: The cache is not persistent across **clearmake** sessions. Each recursive or individual invocation of **clearmake** attempts to contact a view whose tag may have been cached in a previous invocation.

The default time-out period is 60 minutes. To specify a different time-out period, set CCASE_DNVW_RETRY to another integer value (representing minutes). To disable the cache, set CCASE_DNVW_RETRY to 0.

Specifying Views That Can Wink In Derived Objects

You can use the CCASE_WINKIN_VIEWS environment variable to specify a list of views that can wink in derived objects. If this variable is set in the environment or in the makefile, **clearmake** winks in only derived objects that were built in the specified views. If no derived objects are available, **clearmake** rebuilds in the current view.

Specifying a Derived Object in Commands

In general, you use standard pathnames to access DOs when you're working in a view that references them. To standard software (for example, linkers and debuggers), the standard pathname of a derived object (util.o) references the DO.

This is another example of ClearCase transparency: a standard pathname accesses one of many different variants of a file system object. Note this distinction, however:

- A version of an element appears in a dynamic view because it is selected by a configuration specification rule.
- A particular derived object appears in a dynamic view as the result of a build or a winkin.

To access a DO in another dynamic view, use a view-extended pathname:

```
/view/drp/vobs/proj/src/msg.o          (the DO in view drp)
/view/R2_integ/vobs/proj/src/msg.o     (the DO in view R2_integ)
```

Note: You cannot use view-extended pathnames in makefiles.

To specify a certain DO in a ClearCase command, use the DO-ID. For example, you can use a DO-ID in the **catcr** command to view the contents of a specific DO's config record:

cleartool catcr x.o@@29-Jun.14:40.88

Note: You cannot use a DO-ID in standard commands.

Because DO-IDs can change, avoid using them in files or scripts that operate on a DO. Instead, use a standard pathname or the derived object's object identifier (OID), which never changes. To determine a DO's object identifier, use **cleartool describe –fmt "%On\n"**. For example:

cleartool describe –fmt "%On\n" x.o@@29-Jun.14:40.88
```
2c5fc68a.2e5311d3.a382.00:01:80:7b:09:69
```

Also, a derived object is assigned a permanent identifier when it is checked in as a version of an element. See *Working with DO Versions* on page 57.

Winking In a DO Manually

You can manually wink in any DO to your view, using the **winkin** command. For example:

cleartool lsdo hello

```
08-Mar.12:48   akp        "hello@@08-Mar.12:48.265"
07-Mar.11:40   george     "hello@@07-Mar.11:40.217"
```

cleartool winkin hello@@07-Mar.11:40.217

```
Winked in derived object "hello"
```

% hello

```
Greetings, susan!
Your home directory is /net/neptune/home/susan.
It is now Tue Mar  8 12:58:30 2003.
```

You can wink in a DO that does not match your build configuration for any of the following reasons: to run it, to perform a byte-by-byte comparison with another DO, or to perform any other operation that requires access to the DO's file system data.

The **winkin** command can also wink in the set of DOs in a hierarchy of CRs. You can use this recursive winkin to seed a new view with a set of derived objects. For example:

cleartool winkin -recurse x@@20-Jul.14:32.146

```
Winked in derived object "/vobs/smg_test/file2.txt"
Winked in derived object "/vobs/smg_test/file2sub.txt"
Promoting unshared derived object "/vobs/smg_test/x".
Winked in derived object "/vobs/smg_test/x"
```

You can use the **winkin** command to convert a nonshareable DO to a shared DO. For more information, see *Converting Nonshareable DOs to Shared DOs* on page 62.

Preventing Winkin

The following sections describe how to prevent winkin to or from your view during a build or audit. (You can prevent winkins altogether by building in a snapshot view or by not using the ClearCase build tools.)

Preventing Winkin to Your View

To direct **clearmake** to limit reuse to DOs created in the current view, use **clearmake –V**. For more information, see the **clearmake** reference page.

Preventing Winkin to Other Views

To prevent any derived objects that you create from being winked in to other views, use one of the following techniques:

- Use express builds. See *Using Express Builds to Prevent Winkin to Other Views*.
- Use the **–T** or **–F** options to create view-private files with no configuration records. **clearmake** does not perform configuration lookup, but this does not matter if you are not changing other files.
- Use special targets that prevent winkin. For example, use **.NO_WINK_IN** with **clearmake**. For more information, see *Special Targets* on page 77.

Using Express Builds to Prevent Winkin to Other Views

During an express build, Rational ClearCase creates DOs that are nonshareable and cannot be used by builds in other views. These nonshareable DOs have configuration records, but the ClearCase build tools do not write information into the VOB for these DOs. Therefore, the DOs are invisible to builds in other views.

Note: During an express build, the ClearCase build tools wink in DOs from other views. For information about avoiding winkins from other views, see *Preventing Winkin to Your View* on page 54.

Use express builds when DOs created by the build are not appropriate for use by other views. As a general rule:

- Use express builds for development builds that use relatively unstable, checked-out versions.
- Use regular builds for release or nightly builds that use stable, checked-in versions. DOs created by these builds are more likely to be winked in by other views.

Enabling Express Builds

When you invoke a ClearCase build tool, the kind of build that occurs depends on how your view is configured. To use express builds, configure an existing dynamic view with the nonshareable DOs property or create a new dynamic view with the nonshareable DOs property. Then, run your ClearCase build tool (**clearmake** or **clearaudit**) in the view.

The following sections describe how to configure your view to use express builds.

Configuring an Existing View for Express Builds

Use the command **chview –nshareable_dos** *view-tag*. For more information, see the **chview** reference page.

Future builds in the view will create nonshareable derived objects. However, existing DOs in the view are shareable; they are not converted to nonshareable. These existing DOs can still be winked in by other views.

Creating a New View That Uses Express Builds

To create a new view, enter the **mkview** command and specify the **–nshareable_dos** option. For more information, see the **mkview** reference page.

Note: The ClearCase administrator can set a site-wide default for the DO property. If a site-wide default exists and you do not specify the DO property when you create the view, ClearCase uses the site-wide default. For more information, see the **setsite** reference page.

Preventing Winkin to or from Other Architectures

By default, **clearmake** winks in derived objects built on different architectures. For example, a build on a Sun host may wink in a DO built on a Windows NT host. If you want to prevent this behavior, use one of the following techniques:

- Differentiate the build script for different architectures:
  - Add the architecture name to your build script so that **clearmake** differentiates among the build scripts. For example, include `echo $CPU`.
  - Store the architecture name in a macro and pass it to **clearmake** on the command line.
  - Use architecture-specific subdirectories to store DOs.
- Make your tools a build dependency by storing them in a VOB. Also, store your system header files in a VOB.

Converting Derived Objects to View-Private Files

Using a standard command or program to modify a derived object in any way converts it from a DO to a view-private file. For example, use **ls –long** to list a derived object:

```
% cleartool ls –long msg.o
derived object (shared)     msg.o@@10-Mar.15:33.333
```

Modify the DO with a standard command:

```
% touch msg.o
```

The **ls –long** command now lists the file as a view-private object:

```
% cleartool ls –long msg.o
view private object     msg.o
```

Working with DO Versions

The following sections describe how to create and manipulate DO versions.

Creating DO Versions

You can convert DOs to elements or check them in as versions of existing elements. The element-creation and version-creation processes are the same for shareable and nonshareable DOs, with this exception: when you check in a nonshareable DO, it is converted to a shared DO before being checked in.

For more information, see the **mkelem** and **checkin** reference pages.

Checking In DOs During a Build

You can write a build script that creates a derived object and then checks it in or converts it to an element. However, the ClearCase build tool does not create a configuration record until the build script has completed (all commands after the *target-name*: have executed). Therefore, if the same build script that created the DO checks it in or converts it to an element, the resulting version is not a DO version.

For example, the version created by the following build script is not a DO version:

```
buildit : buildit.c
       cleartool co –unres –nc $@
       rm –fr $@
       cc –o $@ $*.c
       cleartool ci –nc buildit
```

You can work around this problem by building and checking in a derived object in two steps. For example, the makefile contains one build script that creates the DO, and another build script that checks it in, as shown here:

```
buildit : buildit.c
       cleartool co –unres –nc $@
       rm –fr $@
       cc –o $@ $*.c

stageit : buildit
       cleartool ci –nc buildit
```

The command **clearmake stageit** performs the following steps:

1 Brings the target buildit up to date. This creates a DO named buildit and an associated configuration record.

2 Brings the target stageit up to date. This step checks in the buildit derived object as a DO version.

Accessing DO Versions

When you check out a DO version, it is winked in to your dynamic view. You can use a standard pathname to access the DO's file system data. However, VOB-database access is handled in the following ways:

- A standard pathname to the DO references the version in the VOB database from which the checkout was made:

```
% cleartool checkout -nc hello                    (wink in derived object hello)
Checked out "hello" from version "/main/3".
```

```
% cleartool mklabel EXPER hello                   (use standard pathname to
Created label "EXPER" on "hello" version          access version from which
"/main/3".                                        checkout was made)
```

- To access the checked-out placeholder version, you must use an extended pathname:

```
% cleartool mklabel -replace EXPER hello@@/main/CHECKEDOUT
Moved label "EXPER" on "hello" from version "/main/3" to
"/main/CHECKEDOUT".
```

If you process a checked-out DO version, as described in *Converting Derived Objects to View-Private Files* on page 56, ClearCase reverts to its usual handling of checked-out versions. In this case, a standard pathname references the placeholder version in the VOB database.

Displaying Configuration Records for DO Versions

The **catcr** command displays the configuration record for a DO version. When you use **catcr –recurse** to display the CRs for a DO and all its subtargets, it does not display the CRs for DO versions unless you use the **–ci** option.

catcr allows precise control over report contents and format. It includes input and output filters and supports a variety of report styles. Input filters, such as **–select**, control which DOs are evaluated. All DOs that are evaluated can potentially appear in the final listing. Output filters, such as **–view_only**, control which DOs actually appear in the final listing. Often, this is a subset of all evaluated DOs.

You can tailor the report in several ways:

- Generate a separate report for each derived object on the command line (default), or a single, composite report for all derived objects on the command line (**–union**).
- Specify which derived objects to consider when compiling report output. The **–recurse**, **–flat**, **–union**, **–ci**, and **–select** options control which subtargets are evaluated. They generate recursive or flat-recursive reports of subtargets, evaluate checked-in DOs, and allow you to evaluate DOs with a particular name only.
- Select the kinds of items that appear in the report. The **–element_only**, **–view_only**, **–type**, **–name**, and **–critical_only** options exclude certain items from the report.
- Display the CR in makefile format (**–makefile**), rather than in a section-oriented format.
- Choose a normal, long, or short report style. Expanding the listing with **–long** adds comments and supplementary information; restricting the listing with **–short** lists file system objects only. You can also list simple pathnames rather than version-extended pathnames (**–nxname**), and relative pathnames rather than full pathnames (**–wd**).

The **–check** option determines whether the CR contains any unusual entries. For example, it determines whether the CR contains multiple versions of the same element, or multiple references to the same element with different names.

By default, **catcr** suppresses a CR entirely if the specified filters remove all objects (useful for searching). With the **–zero** option, the listing includes the headers of such CRs.

The following examples show how to display configuration records for DO versions.

- To display the configuration record for a single DO version:

cleartool catcr x

```
Target x built by smg.user
Host "radar" running SunOS 5.5.1 (sun4m)
Reference Time 20-Jul-03.14:32:32, this audit started
20-Jul-03.14:32:32
View was radar:/home/smg/views/smg_test.vws
Initial working directory was /vobs/smg_test
----------------------------
MVFS objects:
----------------------------
/vobs/smg_test/file1.txt@@/main/2             <30-Jun-03.15:35:12>
/vobs/smg_test/file2.txt@@30-Jun.15:33.115
/vobs/smg_test/x@@/main/3
----------------------------
Build Script:
----------------------------
        cat file1.txt > x
        cat file2.txt >> x
----------------------------
```

- To display the configuration record for a derived object, including the CRs of all subtargets except DO versions:

  cleartool catcr –recurse x

```
----------------------------
----------------------------
Target x built by smg.user
...
Target file2.txt built by smg.user
...
Target file2sub.txt built by smg.user
...
```

- To display the configuration record for a derived object, including the CRs of all subtargets:

cleartool catcr –recurse –ci x

```
------------------------------
------------------------------
Target x built by smg.user
...
Target file1.txt built by smg.user
...
Target file1sub.txt built by smg.user
...
Target file2.txt built by smg.user
...
Target file2sub.txt built by smg.user
...
```

DOs in Unavailable Views

catcr maintains a cache of tags of inaccessible views. For each view tag, the command records the time of the first unsuccessful contact. Before trying to access a view, the command checks the cache. If the view's tag is not listed in the cache, the command tries to contact the view. If the view's tag is listed in the cache, the command compares the time elapsed since the last attempt with the time-out period specified by the CCASE_DNVW_RETRY environment variable. If the elapsed time is greater than the time-out period, the command removes the view tag from the cache and tries to contact the view again.

The default time-out period is 60 minutes. To specify a different time-out period, set CCASE_DNVW_RETRY to another integer value (representing minutes). To disable the cache, set CCASE_DNVW_RETRY to 0.

For more information, see the **catcr** reference page.

Releasing DOs

A project team can use DO versions to make its product (for example, a library) available to other teams. Typically, the team establishes a release area in a separate VOB. For example:

- A library is built by its project team in one location—perhaps /vobs/monet/lib/libmonet.a.
- The team periodically releases the library by creating a new version of a publicly accessible element—perhaps /vobs/publib/libmonet.a.

You can generalize the idea of maintaining a development release area to maintaining a product release area. For example, a Release Engineering group maintains one or more release tree VOBs. The directory structure of the trees mirrors the hierarchy of

files to be created on the release medium. (Because a release tree involves directory elements, it is easy to change its structure from release to release.) A release tree can be used to organize Release 2.4.3 as follows:

1. When an executable or other file is ready to be released, a release engineer checks it in as a version of an element in the release tree.
2. An appropriate version label (for example, **REL2.4.3**) is attached to that version, either manually by the engineer or automatically with a trigger.
3. When all files to be shipped have been labeled in this way, a release engineer configures a view to select only versions with that version label. As seen through this view, the release tree contains exactly the set of files to be released.
4. To cut a release tape, the engineer issues a command to copy the appropriately configured release tree.

Converting Nonshareable DOs to Shared DOs

Note: You cannot convert a shared or unshared DO to a nonshareable DO.

To convert a nonshareable DO to a shared DO, use the **winkin** command. **winkin** advertises the DO by making it shareable and writing information into the VOB and then promotes it (makes it shared). The command also advertises the DO's sub-DOs and siblings, even if you did not specify the **–siblings** option. This process changes the DO ID for each derived object.

The **view_scrubber –p** command performs the same operation. See the **winkin** and **view_scrubber** reference pages.

Automatic Conversion of Nonshareable DOs to Shareable DOs

Because you can change a view's DO property and shareable DOs cannot have nonshareable sub-DOs or siblings, situations can occur in which **clearmake** must convert nonshareable DOs into shareable DOs.

For example, you set your view's DO property to nonshareable and perform a build, creating nonshareable DOs. You then set your view's DO property to shareable and perform another build. The build tool determines that it can reuse some of the nonshareable DOs created in the first build to create shareable DOs in the second. It converts the nonshareable DOs to shareable DOs and reuses them.

Creating Links to Derived Objects

You cannot make a VOB hard link to a derived object. You can make one or more view-private hard links to a derived object, using the UNIX **ln** command, with these restrictions:

- The derived object must be visible in the dynamic view where the view-private hard link is to be created; that is, it must appear in a standard UNIX **ls** listing. (You can use the **winkin** command to satisfy this requirement.)
- The pathname of the hard link must be within the same VOB as the original derived object.

All hard links to a derived object, including the name under which it was originally created, appear with the same DO ID in a ClearCase **ls** listing; if there are multiple names for a derived object in the same directory, all are listed. For example:

```
% ln hello hw
% cleartool ls
...
hello@@19-May.19:15.232
hw@@19-May.19:15.232
...
```

In a **catcr** or **describe** command, you can reference a derived object using any of its hard links; all the references are equivalent. But an **lsdo** command must reference a derived object by its original name, not by any of its subsequently created hard links. Likewise, a derived object can be winked in only at its original pathname.

Special case: If a hard link is created by the same build script as the derived object itself, the hard link becomes an additional name for the DO. **lsdo** lists the hard link, and **clearmake** can wink in using the hard link's pathname.

Each additional hard link increments a derived object's reference count. An **lsdo –l** listing includes the reference counts and the dynamic views in which the references exist. The `(2)` in this example shows that view **old.vws** has two references to hello:

```
% cleartool lsdo -long hello
08-Dec-03.12:06:19 Chuck Jackson (test user) (jackson.dvt@oxygen)
 create derived object "hello@@08-Dec.12:06.234"
 references: 2 => oxygen:/usr/vobstore/tut/old.vws (2)
```

Displaying VOB Disk Space Used for Derived Objects

The **dospace** command reports VOB disk space used for shared derived objects. For more information, see the **dospace** reference page and the *Administrator's Guide* for Rational ClearCase.

Deleting Derived Objects

The **rmdo** command removes the data container and the VOB database object for a derived object. For more information, see the **rmdo** reference page.

Shareable derived objects and their data containers can be deleted independently. Deleting a nonshareable derived object deletes the DO.

Removing Data Containers for Derived Objects

The standard **rm(1)** command causes a shareable derived object to disappear from the dynamic view. The effect on physical data storage is as follows:

- If the DO's data container is in the view's private storage area, **rm** deletes that data container.
- If the DO's data container is in a VOB storage pool, the data container is not affected.

In both cases, the derived object in the VOB database is not deleted. The only change to the derived object is that its reference count is decremented.

When a build overwrites a nonshareable or unshared DO, the MVFS removes the old data container from the dynamic view's private storage area, and creates a new one there. It also creates a new CR. At the operating system level, the effect is that an existing file is overwritten.

Scrubbing Derived Objects and Data Containers

A reference count of zero means that the derived object has been deleted or overwritten in every view that ever used it. This situation calls for *scrubbing*: automatic deletion of DO-related information from the VOB. Scrubbing can remove the derived object from the VOB database, its data container from a VOB storage pool (if the DO had ever been shared), and in some cases its associated CR, as well.

The **scrubber** utility removes derived objects from a VOB database and data containers from VOB storage pools. The **view_scrubber** utility removes data containers from a

dynamic view's private storage area. For more information about scrubbing, see the *Administrator's Guide* for Rational ClearCase.

Degenerate Derived Objects

A derived object is complete if its VOB database object, data container, and configuration record (CR) are accessible. Because these entities exist independently, a derived object can become incomplete, or degenerate, if one entity is missing.

Data Container Deleted

When an unshared DO is removed with **rm** or by a target rebuild, its VOB database object continues to exist in the VOB database (with a zero reference count), but the data container no longer exists. Such DOs are usually ignored by **lsdo**, but can be listed with the **–zero** option. The **scrubber** utility deletes zero-referenced DOs.

The **checkvob** command can find and fix missing container problems.

DO Deleted from VOB Database

When an unshared DO is removed from its VOB database with **rmdo**, the data container continues to be visible:

```
% cleartool rmdo Vhelp.log
Removed derived object "Vhelp.log@@14-Sep.72783".
% cleartool ls Vhelp.log
Vhelp.log [no config record]
```

In general, try to avoid using the **rmdo** command.

CR Unavailable

A newly created CR is stored in the dynamic view where its associated DOs were built. If that view becomes unavailable (for example, it is inadvertently destroyed or its host is temporarily down), the DO continues to exist in the VOB database, but operations that must access the CR fail:

```
cleartool: Error: Unable to find view 'mars:/viewstore/pink.vws'
from albd: error detected by ClearCase subsystem
cleartool: Error: See albd_log on host mars
cleartool: Error: Unable to contact View - error detected by ClearCase
subsystem
```

Displaying Contents of Configuration Records

The **catcr** command displays the contents of a configuration record. For more information, see the **catcr** reference page.

Comparing Configuration Records

Because config records provide complete records of how DOs are built, you can use them to determine how two builds differ. For example, you expected the build to reuse or wink in a DO that it rebuilt instead. You can compare the CRs for the two DOs to find out what aspect of the build environment was different.

To compare two existing CRs, use the **diffcr** command. For more information, see the **diffcr** reference page.

Attaching Labels or Attributes to Versions in a CR

You can attach a label or an attribute to the versions in the CR hierarchy of a derived object.

For example, to attach the **SMG_BUILD_5_03** label to the versions in the CR hierarchy of file.o:

cleartool mklabel –c "may 03 build" –config file.o SMG_BUILD_5_03

```
Created label "SMG_BUILD_5_03" on "/vobs/smg_test/" version
"/main/CHECKEDOUT".
Created label "SMG_BUILD_5_03" on "/vobs/smg_test/acc.c" version
"/main/2".
Created label "SMG_BUILD_5_03" on "/vobs/smg_test/file.c" version
"/main/1".
```

For more information, see the description of the **–config** option in the **mkattr** and **mklabel** reference pages.

Configuring a View to Select Versions Used to Build a DO

To select the versions in the CR hierarchy of a derived object, use the **–config** version selector in your view's config spec. For example, the following config spec selects the versions in the CR hierarchy for hello.o:

```
element * CHECKEDOUT
element * –config /vobs/dev/lib/hello.o
element * /main/v3.8/LATEST
```

For more information, see the **config_spec** reference page.

Including a Makefile Version in a Configuration Record

To record a makefile version in a CR, use one of the following methods:

- Declare it as an explicit dependency in the makefile. To do this, you can use the **$(MAKEFILE)** variable. You must explicitly list any included makefiles that you want to record.

  The drawback to this method is that it causes targets that depend on the makefile to be rebuilt if there is any change to the makefile.

- Make it an implicit dependency by referring to it in a build script and use the special target **.DEPENDENCY_IGNORED_FOR_REUSE** to ignore it in subsequent rebuild decisions. You must explicitly list any included makefiles that you want to record.

  For example:

  ```
  .DEPENDENCY_IGNORED_FOR_REUSE: $(MAKEFILE)
  targ: dep1 dep2
     cat $(MAKEFILE) > /dev/null
     touch targ
  ```

  The drawback to this method is that the makefile dependency is ignored for reuse, but it is not ignored for winkin.

- Use the **.MAKEFILES_IN_CONFIG_REC** special target. See *Special Targets* on page 77.

clearmake Makefiles and BOS Files

5

This chapter describes makefiles processed by the Rational ClearCase build program **clearmake**. This is a discussion of differences and ClearCase extensions rather than a complete description of makefile syntax. This chapter also describes build option specification files (BOS files), which contain temporary macros and ClearCase special targets.

Makefile Overview

A makefile contains a sequence of entries, each of which specifies a build target, some dependencies, and the build scripts of commands to be executed. A makefile can also contain **make** macro definitions, target-dependent macro definitions, and build directives (special targets.)

- **Target/dependencies line**. The first line of an entry is a white-space-separated, nonnull list of targets, followed by a colon (**:**) or a double colon (**::**), and a (possibly empty) list of dependencies. Both targets and dependencies may contain ClearCase pathname patterns. (See the **wildcards_ccase** reference page.)

  The list of dependencies may not need to include source objects, such as header files, because **clearmake** detects these dependencies. However, the list must include build-order dependencies, for example, object modules and libraries that must be built before executables. (See *Build-Order Dependencies* on page 36.)

- **Build script**. Text that follows a semicolon (**;**) on the same line and all subsequent lines that begin with a <TAB> character constitute a build script: a set of shell commands to be executed. A shell command can be continued onto the next text line with a \<NL> sequence. Any line beginning with a number sign (**#**) is a comment.

  A build script ends at the first nonempty line that does not begin with a <TAB> or number sign (**#**); this begins a new target/dependencies line or a make macro definition.

  Build scripts must use standard pathnames only. Do not include view-extended or version-extended pathnames in a build script.

Executing a build script updates the target and is called a *target rebuild*. The shell commands in a build script are executed one at a time, each in its own instances of the subshell.

Note that **clearmake** always completely eliminates a \<NL> sequence, even in its compatibility modes. Some other **make** programs sometimes preserve such a sequence—for example, in a **sed(1)** insert command:

```
target: depdcy

sed -e '/xxx=0/i\

yyy=xxx;' depdcy > target
```

- **Make macro**. A **make** macro is an assignment of a character-string value to a simple name. By convention, all letters in the name are uppercase (for example, **CFLAGS**).
- **Target-dependent macro definitions**. A target-dependent macro definition takes the form *target-list* **:=** *macro_name* **=** *string*

  You can use macros in makefiles or in BOS files. For more information, see *Target-Dependent Macro Definitions* on page 84.
- **Special targets**. A line that begins with a dot (**.**) is a special target, which acts as a directive to **clearmake**.
- **Special characters in target names**. You can use special characters in target names by immediately preceding each special character with a backslash (\).

Build Options Specification Files

A build options specification (BOS) file is a text file containing macro definitions and/or ClearCase special targets. We recommend that you place temporary macros (such as **CFLAGS=–g** and others not to be included in a makefile permanently) in a BOS file, rather than specifying them on the **clearmake** command line.

By default, **clearmake** reads BOS files in this order:

1. The default BOS files
   a. The file .clearmake.options in your home directory (as indicated in the password database), which is the place for macros to be used every time you execute **clearmake**.
   b. One or more local BOS files, each of which corresponds to one of the makefiles specified with a **–f** option or read by **clearmake**. Each BOS file has a name in the form makefile-name.options. For example:

```
makefile.options

Makefile.options

project.mk.options
```

2 BOS files specified in the CCASE_OPTS_SPECS environment variable.

3 BOS files specified on the command line with **–A**.

If you specify **–N**, **clearmake** does not read default BOS files.

clearmake displays the names of the BOS files it reads if you specify the **–v** or **–d** option, or if %CCASE_VERBOSITY >= 1.

For information about the contents of BOS files, see *Setting Up the Client Host* on page 143.

When **clearmake** shops for a derived object to wink in to a build, it may find DOs from a view that is unavailable (because the view server host is down, the **albd_server** is not running on the server host, and so on). Attempting to fetch the DO's configuration record from an unavailable view causes a long time-out, and the build may reference multiple DOs from the same view.

clearmake and other **cleartool** commands that access configuration records and DOs (**lsdo**, **describe**, **catcr**, **diffcr**) maintain a cache of tags for inaccessible views. For each view tag, the command records the time of the first unsuccessful contact. Before trying to access a view, the command checks the cache. If the view's tag is not listed in the cache, the command tries to contact the view. If the view's tag is listed in the cache, the command compares the time elapsed since the last attempt with the time-out period specified by the CCASE_DNVW_RETRY environment variable. If the elapsed time is greater than the time-out period, the command removes the view tag from the cache and tries to contact the view again.

Note: The cache is not persistent across **clearmake** sessions. Each recursive or individual invocation of **clearmake** attempts to contact a view whose tag may have been cached in a previous invocation.

The default time-out period is 60 minutes. To specify a different time-out period, set CCASE_DNVW_RETRY to another integer value (representing minutes). To disable the cache, set CCASE_DNVW_RETRY to 0.

Format of Makefiles

The following sections describe the special considerations for using makefiles with **clearmake**.

Note: For information about environment variables that affect clearmake, see the **env_ccase** reference page. You can also use the **-d** or **-v** option to the **clearmake** command to view a list of environment variables that **clearmake** reads during the build.

Restrictions

clearmake does not support the use of standard input as a makefile.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive (library) created by **ar(1)**. For example:

```
lib.a : lib.a(mod1.o) lib.a(mod2.o)
```

The string within parentheses refers to a member (object module) within the library. Use of function names within parentheses is not supported. Thus, **lib.a(mod1.o)** refers to an archive that contains object module **mod1.o**. The expression **lib.a(mod1.o mod2.o)** is not valid.

Inference rules for archive libraries have this form:

.*sfx*.a
: where *sfx* is the file name extension (suffix) from which the archive member is to be made.

The way in which **clearmake** handles incremental archive construction differs from other **make** variants. For more information, see *Working with Incremental Update Tools* on page 42.

Note: The **u** key for **ar** is not reliable within a ClearCase environment. Do not use it.

Command Echoing and Error Handling

You can control the echoing of commands and the handling of errors that occur during command execution on a line-by-line basis or on a global basis.

You can prefix any command with one or two characters, as follows:

\–
: Causes **clearmake** to ignore any errors during execution of the command. By default, an error causes **clearmake** to terminate.

  The command-line option **–i** suppresses termination-on-error for all command lines.

@	Suppresses display of the command line. By default, **clearmake** displays each command line just before executing it. The command-line option **–s** suppresses display of all command lines. The **–n** option displays commands, but does not execute them.
–@ @–	These two prefixes combine the effect of **–** and **@**.

The **–k** option provides for partial recovery from errors. If an error occurs, execution of the current target (that is, the set of commands for the current target) stops, but execution continues on other targets that do not depend on that target.

clearmake reserves **-2 (254)** as a return code meaning "unable to execute." This error is distinct from shell-returned errors and cannot be ignored by the use of **-i**. Users who plan to write scripts that return standard error codes should avoid the use of **-2 (254)**.

Built-In Rules

File name extensions (suffixes) and their associated rules in the makefile override any identical file name extensions in the built-in rules. **clearmake** reads built-in rules from the file *ccase-home-dir*/etc/builtin.mk when you run in standard compatibility mode. In other compatibility modes, other files are read.

Include Files

If a line in a makefile starts with the string **include** or **sinclude** followed by white space (at least one <SPACE> or <TAB> character), the rest of the line is assumed to be a file name. (This name can contain macros.) The contents of the file are placed at the current location in the makefile.

For **include**, a fatal error occurs if the file is not readable. For **sinclude**, a nonreadable file is silently ignored.

Macros

The following sections describe the order of precedence of macros in a **clearmake** build, and the different types of macros.

Order of Precedence of Make Macros and Environment Variables

By default, the order of precedence of macros and environment variables is as follows:

1. Target-dependent macro definitions
2. Macros specified on the **clearmake** command line
3. Make macros set in a BOS file

4 Make macro definitions in a makefile

5 Environment variables

For example, target-dependent macro definitions override all other macro definitions, and macros specified on the **clearmake** command line override those set in a BOS file.

If you use the **–e** option to **clearmake**, environment variables override macro definitions in the makefile.

All BOS file macros (except those overridden on the command line) are placed in the build script's environment. If a build script recursively invokes **clearmake**:

- The higher-level BOS file setting (now transformed into an EV) is overridden by a make macro set in the lower-level makefile. However, if the recursive invocation uses **clearmake**'s **–e** option, the BOS file setting prevails.
- If another BOS file (associated with another makefile) is read at the lower level, its make macros override those from the higher-level BOS file.

For a list of all environment variables, see the **env_ccase** reference page.

Make Macros

A *macro definition* takes this form:

macro_name **=** *string*

Macros can appear in the makefile, on the command line, or in a build options specification file. (See *Build Options Specification Files* on page 70.)

Macro definitions require no quotes or delimiters, except for the equal sign (=), which separates the macro name from the value. Leading and trailing white-space characters are stripped. Lines can be continued using a \<NL> sequence; this sequence and all surrounding white space is effectively converted to a single <SPACE> character. *macro_name* cannot include white space, but *string* can; it includes all characters up to an unescaped <NL> character.

clearmake performs macro substitution whenever it encounters either of the following in the makefile:

$(*macro_name***)**
$(*macro_name***:***subst1***=***subst2***)**

It substitutes *string* for the macro invocation. In the latter form, **clearmake** performs an additional substitution within *string*: all occurrences of *subst1* at the end of a word within *string* are replaced by *subst2*. If *subst1* is empty, *subst2* is appended to each word in the value of *macro_name*. If *subst2* is empty, *subst1* is removed from each word in the value of *macro_name*.

For example:

```
% cat Makefile
C_SOURCES = one.c two.c three.c four.c
test:
     echo "OBJECT FILES are: $(C_SOURCES:.c=.o)"
     echo "EXECUTABLES are: $(C_SOURCES:.c=)"
% clearmake test
OBJECT FILES are: one.o two.o three.o four.o
EXECUTABLES are: one two three four
```

Internal Macros

clearmake maintains these macros internally. They are useful in rules for building targets.

$*	(Defined only for inference rules) The file name part of the inferred dependency, with the file name extension deleted.
$@	The full target name of the current target.
$<	(Defined only for inference rules) The file name of the implicit dependency.
$?	(Defined only when explicit rules from the makefile are evaluated) The list of dependencies that are out of date with respect to the target. When configuration lookup is enabled (default), it expands to the list of all dependencies, unless that behavior is modified with the **.INCREMENTAL_TARGET** special target. In that case, **$?** expands to the list of all dependencies different from the previously recorded versions. When a dependency is an archive library member of the form lib(file.o), the name of the member, file.o, appears in the list.
$%	(Defined only when the target is an archive library member) For a target of the form lib(file.o), **$@** evaluates to `lib` and **$%** evaluates to the library member, file.o.
MAKE	The name of the make processor (that is, **clearmake**). This macro is useful for recursive invocation of **clearmake**.

MAKEFILE During makefile parsing, this macro expands to the pathname of the current makefile. After makefile parsing is complete, it expands to the pathname of the last makefile that was parsed. This holds only for top-level makefiles, not for included makefiles or for built-in rules; in these cases, it echoes the name of the including makefile.

Use this macro as an explicit dependency to include the version of the makefile in the CR produced by a target rebuild. For example:

```
supersort: main.o sort.o cmd.o $(MAKEFILE)
    cc -o supersort ...
```

For more information, see *Including a Makefile Version in a Configuration Record* on page 67.

MAKEFLAGS This macro passes flags to sub-makes, including flags that take arguments and macro definitions. **clearmake** reads the contents of **MAKEFLAGS** at startup and amends it to include any flags not specific to ClearCase passed at the command line. Any flags specific to ClearCase are passed through **CCASE_MAKEFLAGS** and if **clearmake** detects these flags in **MAKEFLAGS**, it moves them to **CCASE_MAKEFLAGS**.

Flags passed through **MAKEFLAGS**:

-I, **-p**, **-N**, **-w**, **-e**, **-r**, **-i**, **-k**, **-n**, **-q**, **-s**

Flags passed through **CCASE_MAKEFLAGS**:

-A, **-B**, **-N**, **-b**, **-v**, **-C**, **-U**, **-M**, **-V**, **-O**, **-T**, **-F**, **-R**, **-c**, **-u**, **-d**

This functionality is available for all compatibility modes.

VPATH Macro

The **VPATH** macro specifies a search path for targets and dependencies. **clearmake** searches directories in **VPATH** when it fails to find a target or dependency in the current working directory. **clearmake** searches only in the current view. The value of **VPATH** can be one directory pathname, or a colon-separated list of directory pathnames. (In Gnu compatibility mode, you can also use spaces as separators.)

As **clearmake** qualifies makefile dependencies (explicit dependencies in the makefile), the process of configuration lookup is **VPATH**-sensitive. Thus, if a newer version of a dependent file appears in a directory on the search path before the pathname in the CR (the version used in the previous build), **clearmake** rejects the previous build and rebuilds the target with the new file.

The **VPATH** setting may affect the expansion of internal macros, such as **$<**.

Special Targets

Like other build tools, **clearmake** interprets certain target names as declarations. Some of these special targets accept lists of patterns as their dependents, as noted in the description of the target. Pattern lists may contain the pattern character, **%**. When evaluating whether a name matches a pattern, the tail of the prefix of the name (subtracting directory names as appropriate) must match the part of the pattern before the %; the file name extension of the name must match the part of the pattern after the %. For example:

Name	Matches	Does not match

`/dir/subdir/x.o`	`%.o`
	`x.o`
	`subdir/%.o`
	`subdir/x.o`
		`/dir/subdir/otherdir/x.o`

The following targets accept lists of patterns:

- **.DEPENDENCY_IGNORED_FOR_REUSE**
- **.INCREMENTAL_REPOSITORY_SIBLING**
- **.INCREMENTAL_TARGET**
- **.NO_CMP_NON_MF_DEPS**
- **.NO_CMP_SCRIPT**
- **.NO_CONFIG_REC**
- **.NO_DO_FOR_SIBLING**
- **.NO_WINK_IN**
- **.SIBLING_IGNORED_FOR_REUSE**

Special Targets for Use in Makefiles

.DEFAULT :
If a file must be built, but there are no explicit commands or relevant built-in rules to build it, the commands associated with this target are used (if it exists).

.IGNORE :
Same effect as the **–i** option.

.PRECIOUS : *tgt ...*
The specified targets are not removed when a quit character (typically, CTRL+\) or an interrupt character (typically, CTRL+C) is typed.

.SILENT :
Same effect as the **–s** option.

Special Targets for Use in Makefiles or BOS Files

You can use the following special targets either in the makefile itself or in a build options specification file. See *Build Options Specification Files* on page 70.

.DEPENDENCY_IGNORED_FOR_REUSE: *file ...*

The dependencies you specify are ignored when **clearmake** determines whether a target object in a VOB is up to date and can be reused. By default, **clearmake** considers that a target cannot be reused if its dependencies have been modified or deleted since it was built. This target applies only to reuse, not to winkin. Also, this target applies only to detected dependencies, which are not declared explicitly in the makefile.

You can specify the list of files with a tail-matching pattern, for example, **Templates.DB/%.module**.

Unlike the files listed in most special targets, the files on this list refer to the names of dependencies and not the names of targets. As such, the special target may apply to the dependencies of many targets at once. This special target is most useful when identifying a class of dependencies found in a particular toolset for which common behavior is desired across all targets that have that dependency.

Note: clearmake does not provide a special target to ignore dependencies for winkin because, in general, is it not safe to winkin the partial results of a build. If the siblings are not used by the build tools, then they should probably be deleted by the build script or created outside the VOB. If the siblings are used by the build tools, then their contents may depend on the sequence of builds performed in that view and it is unsafe for **clearmake** to winkin the other build artifacts while ignoring the dependent siblings.

.INCREMENTAL_REPOSITORY_SIBLING: *file ...*

The files listed are incremental repository files created as siblings of a primary target and may contain incomplete configuration information. This special target is useful when a toolset creates an incremental sibling object and you want to prevent clearmake from winking in a primary target that has this sibling.

You can specify the list of files with a tail-matching pattern, for example, **%.pdb**.

Unlike the files listed in most special targets, the files on this list refer to the names of sibling objects and not the names of targets. As such, the special target may apply to the siblings of many targets at once. This special target is most useful when identifying a class of siblings found in a particular toolset for which common behavior is desired across all targets that have that sibling.

Caution: If you have an incremental sibling and you do not use **.INCREMENTAL_REPOSITORY_SIBLING**, you might wink in the incremental sibling inadvertently, which could overwrite the version of the sibling in your view and cause problems.

.INCREMENTAL_TARGET: *tgt ...*
This target performs incremental configuration record merging for the listed targets; in other words, it combines dependency information from instances of this target generated previously with the current build of this target. This special target is most useful when building library archives, because typically only some of the objects going into a library are read each time the library is updated.

You can specify the list of files with a tail-matching pattern; for example, **%.a**.

For information about restructuring a makefile to build incremental archive files, see *Working with Incremental Update Tools* on page 42.

Note: .INCREMENTAL_TARGET applies only to makefile targets built incrementally using a single make rule. Do not use it for the following kinds of files:

- Files built incrementally that are not makefile targets. For example, sibling objects like log files or template repositories.
- Files built incrementally from several different build scripts.

The general guideline is that if you are not building a library with **ar** in a single makefile rule, and you are not building an executable using an incremental linker, do not use **.INCREMENTAL_TARGET**.

.JAVA_TGTS: *file ...*
This special target is used to handle subclasses generated by Java compilers.

In the makefile, any file name that matches the pattern will allow a **$** to be escaped by another **$**. For example, to specify **a$x.class**:

You can specify the list of files with a tail-matching pattern; for example, **%.class:**

```
.JAVA_TGTS: %.class
.java.class:
        javac $<
a$$x.class: a.class
```

Note that **$$** mapping to a single **$** is default behavior in Gnu make compatibility mode. For more information, see the **makefile_gnu** reference page.

.JAVAC:

Use this special target to enable **clearmake** to use heuristics on audits of Java builds to accurately evaluate .class dependencies. These dependencies are then stored in .class.dep files for future **clearmake** runs, and they enable those runs to build .class targets in the same order that the Java compiler does.

This special target must be used with no dependencies and no build script:

```
JAVAC:
```

Other than that, makefiles must use implicit suffix or pattern rules. For example:

```
.SUFFIXES: .java .class

.java.class:
        rm -f $@
        $(JAVAC) $(JFLAGS) $<
```

For compatibility modes that support them, use implicit pattern rules. For example:

```
%.class: %.java
        rm -f $@
        $(JAVAC) $(JFLAGS) $?
```

The makefiles must also use absolute paths for .class targets. For example:

```
all_classes:/vobs/proj/src/pkg1/foo.class
```

clearmake contains a builtin macro function you can use to specify absolute paths:

```
$(javaclasses)
```

For more information about the **.JAVAC** special target, see Chapter 8.

.MAKEFILES_IN_CONFIG_REC: *file ...*

Use this special target to record the versions of makefiles in the configuration records of derived objects.

This target takes an optional dependency list, which may be a pattern. When used without a dependency list, this target causes all makefiles read by a build session to be recorded in the configuration record of all derived objects built during that build session.

To conserve disk space, you may want to supply a dependency list to this target so that, for example, only DOs built for top-level targets have the makefiles recorded in their configuration records.

.MAKEFILES_AFFECT_REUSE:

By default, makefiles recorded by using the **.MAKEFILES_IN_CONFIG_REC** special target do not affect DO reuse. You can use this target to enable recorded makefiles to affect DO reuse. (If you want to have some recorded makefiles affect reuse, but not all, you can also use the **.DEPENDENCY_IGNORED_FOR_REUSE** special target in conjunction with this target.)

Note: If a makefile is declared an explicit dependency of a target, it always affects DO reuse for that target, whether or not .**MAKEFILES_AFFECT_REUSE** was used.

Makefiles recorded in a configuration record are labeled by **mklabel –config**. Makefiles that were recorded in a configuration record but that were not recorded by using .**MAKEFILES_AFFECT_REUSE** are ignored by **catcr –critical_only** and **diffcr –critical_only**.

.NO_CMP_NON_MF_DEPS: *tgt ...*

The specified targets are built as if the **–M** option were specified; if a dependency is not declared in the makefile, it is not used in configuration lookup.

You can specify the list of files with a tail-matching pattern, for example, %.o.

.NO_CMP_SCRIPT : *tgt ...*

The specified targets are built as if the **–O** option were specified; build scripts are not compared during configuration lookup. This is useful when different makefiles (and, hence, different build scripts) are regularly used to build the same target.

You can specify the list of files with a tail-matching pattern, for example, %.o.

.NO_CONFIG_REC : *tgt ...*

The specified targets are built as if the **–F** option were specified; modification time is used for build avoidance, and no CRs or derived objects are created.

You can specify the list of files with a tail-matching pattern; for example, %.o.

.NO_DO_FOR_SIBLING : *file ...*

Use this target to disable the creation of a derived object for any file listed if that file is created as a sibling derived object (an object created by the same build rule that created the target). These sibling derived objects are left as view-private files.

You can specify the list of files with a tail-matching pattern, for example, **ptrepository/_%**.

Unlike the files listed in most special targets, the files on this list refer to the names of sibling objects and not the names of targets. As such, the special target may apply to the siblings of many targets at once. This special target is most useful when identifying a class of siblings found in a particular toolset for which common behavior is desired across all targets that have that sibling.

.NO_WINK_IN : *tgt ...*
The specified targets are built as if the **–V** option were specified; configuration lookup is restricted to the current view.

You can specify the list of files with a tail-matching pattern, for example, %.o.

.NOTPARALLEL : *tgt ...*
Without any *tgt* arguments, disables parallel building for the current makefile. **clearmake** builds the entire makefile serially, one target at a time. With a set of *tgt* arguments, prevents **clearmake** from building any of the targets in the set in parallel with each other. However, targets in a set can be built in parallel with targets in a different set or with any other targets. For example:

```
.NOTPARALLEL:%.a
.NOTPARALLEL:acc1 acc2
```

clearmake does not build any .a file in parallel with any other .a file, and acc1 is not built in parallel with acc2. However, **clearmake** may build .a files in parallel with acc1 or acc2.

.NOTPARALLEL does not affect lower-level builds in a recursive make, unless you specify it in the makefiles for those builds or include it in a BOS file.

You can specify the list of files with a tail-matching pattern, for example, %.a.

See also Chapter 9, *Setting Up a Parallel Build*.

.SIBLING_IGNORED_FOR_REUSE: *file ...*
The *files* are ignored when **clearmake** determines whether a target object in a VOB is up to date and can be reused. This is the default behavior, but this special target can be useful in conjunction with the **.SIBLINGS_AFFECT_REUSE** special target or **–R** command-line option. This target applies only to reuse, not to winkin.

You can specify the list of files with a tail-matching pattern, for example, **Templates.DB/%.module**.

Unlike the files listed in most special targets, the files on this list refer to the names of sibling objects and not the names of targets. As such, the special target may apply to the siblings of many targets at once. This directive is most useful when identifying a class of siblings found in a particular toolset for which common behavior is desired across all targets that have that sibling.

.SIBLINGS_AFFECT_REUSE:

Build as if the **–R** command line option were specified; examine sibling derived objects when determining whether a target object in a VOB can be reused (is up to date). By default, when determining whether a target is up to date, **clearmake** ignores modifications to objects created by the same build rule that created the target (sibling derived objects). This directive tells **clearmake** to consider a target out of date if its siblings have been modified or deleted.

Sharing Makefiles Between UNIX and Windows

clearmake is available on both UNIX and Windows NT. In principle, you can write portable makefiles, but in practice, the obstacles are substantial. The variations in tool and argument names between systems makes writing portable build scripts particularly challenging. If you choose to pursue portable makefiles, use the following general procedures to produce usable results.

- **Start on UNIX; avoid most compatibility modes.** On Windows NT, **clearmake** supports Gnu compatibility mode but does not support others (for example, Sun compatibility mode). Instead, it supports basic **make** syntax. To write or tailor transportable makefiles, begin makefile development on UNIX, without compatibility modes other than Gnu in effect. Gnu generates errors and warnings for problematic syntax. When things work cleanly on UNIX, move your makefiles to Windows NT for testing.
- **Use a makefile-generating utility, such as imake, to generate makefiles**. Use **imake** or some other utility to generate the makefiles you will need, including **clearmake** makefiles for Windows NT.

BOS File Entries

The following sections describe the entries you can put in BOS files.

Standard Macro Definitions

A standard macro definition has the same form as a make macro defined in a makefile:

macro_name **=** *string*

For example:

```
CDEBUGFLAGS = -g
```

Target-Dependent Macro Definitions

A target-dependent macro definition takes this form:

target-pattern-list **:=** *macro_name* **=** *string*

Any standard macro definition can follow the **:=** operator; the definition takes effect only when the targets that match patterns in *target-pattern-list* and their dependencies are processed. Patterns in the *target-pattern-list* must be separated by white space. For example:

```
x.o y.o := CDEBUGFLAGS=-g
```

Two or more higher-level targets can have a common dependency. If the targets have different target-dependent macro definitions, the dependency is built using the macros for the first higher-level target **clearmake** considered building (whether or not **clearmake** actually built it).

Shell Command Macro Definitions

A shell command macro definition replaces a macro name with the output of a shell command:

macro_name **:sh =** *string*

This defines the value of *macro_name* to be the output of *string*, any shell command. In command output, <NL> characters are replaced by <SPACE> characters. For example:

```
BUILD_DATE :sh = date
```

Special Targets

You can use some ClearCase special targets in a build options spec. See *Special Targets for Use in Makefiles or BOS Files* on page 78.

Include Directives

To include one BOS file in another, use the **include** or **sinclude** (silent include) directive. For example:

```
include /usr/local/lib/ux.options
sinclude $(OPTS_DIR)/pm_build.options
```

Comments

A BOS file can contain comment lines, which begin with a number sign (**#**).

Conflict Resolution

Conflicts can occur in specifications of **make** macros and environment variables. For example, the same **make** macro may be specified both in a makefile and on the command line; or the same name may be specified both as a make macro and as an environment variable.

clearmake resolves such conflicts similarly to other **make** variants; it uses the following priority order, from highest to lowest:

1. Target-specific macros specified in a BOS file
2. Target-specific macros specified in a makefile
3. Make macros specified on the command line
4. Make macros specified in a BOS file
5. Make macros specified in a makefile
6. Environment variables
7. Built-in macros

Using the **–e** option gives environment variables higher priority than **make** macros specified in a makefile.

Conflict resolution details. The following discussion treats this topic more precisely but less concisely.

clearmake starts by converting all EVs in its environment to make macros. (SHELL is an exception.) These EVs are also placed in the environment of the shell process in which a build script executes. Then, it adds in the make macros declared in the makefile. If this produces name conflicts, they are resolved as follows:

- If **clearmake** was not invoked with the **–e** option, the macro value overwrites the EV value in the environment.
- If **clearmake** was invoked with the **–e** option, the EV value becomes the value of the make macro.

Finally, **clearmake** adds make macros specified on the command line or in a BOS file; these settings are also added to the environment. These assignments *always* override any others that conflict. (A command-line assignment overrides a BOS setting of the same macro.)

SHELL Environment Variable

clearmake does not use the SHELL environment variable to select the shell program in which to execute build scripts. It uses a UNIX Bourne shell (**/bin/sh**) , unless you specify another program with a **SHELL** macro. You can specify **SHELL** on the command line, in the makefile, or in a build options spec; the value of **SHELL** must be a full pathname.

Note: If **clearmake** determines that it can execute the build script directly, it does not use the shell program even if you specify one explicitly. To force **clearmake** to always use the shell program, set the environment variable CCASE_SHELL_REQUIRED.

CCASE_BRANCH0_REUSE Environment Variable

When **clearmake** evaluates a derived object for usability in the view, it compares versions of files recorded in the derived object's configuration record to versions of those same files that are in the current view. Generally, any version mismatch prevents **clearmake** from reusing the DO. However, if the DO used version main/123 and the view sees version main/123/branch/0, **clearmake** considers this to be a match. You can disable this default behavior by setting the environment variable CCASE_BRANCH0_REUSE in the shell before running **clearmake**.

Using clearmake Compatibility Modes

6

clearmake is designed for compatibility with existing make programs, which minimizes the changes you need to make to your makefiles. There are many variants of **make**, and each provides different sets of extended features. **clearmake** does not support all features of all variants, and we do not guarantee absolute compatibility.

If your makefiles use only the common extensions, they will probably work with **clearmake** without changes. If you must use features that **clearmake** does not support, consider using another make program in a **clearaudit** shell. This alternative provides build auditing (configuration records), but does not provide build avoidance (*winkin*).

Note: When building with configuration records, **clearmake** handles double-colon rules differently from other **make** programs. For more information, see *How clearmake Interprets Double-Colon Rules* on page 36.

To specify a compatibility mode, take one of the following actions:

- Use the environment variable CCASE_MAKE_COMPAT in a build options specification file or in your environment. For more information, see Chapter 5, *clearmake Makefiles and BOS Files*.
- Use the **–C** option with **clearmake**. For more information, see the **clearmake** reference page.

You can use the following compatibility modes:

sgismake	IRIX smake
sgipmake	IRIX pmake
sun	SunOS make
aix	IBM AIX make
gnu	Free Software Foundation Gnu make
std	Standard **clearmake** with no compatibility mode enabled. (Use this option to nullify a setting of the environment variable CCASE_MAKE_COMPAT.)

For information about specific **clearmake** compatibility modes, see the **makefile_aix**, **makefile_pmake**, **makefile_smake**, **makefile_sun**, **makefile_gnu** and **makefile_ccase** reference pages.

Using ClearCase to Build C++ Programs

7

This chapter describes how to use **clearmake** effectively with C++ programs built with various compilers. If you use **clearmake** to build C++ programs, read the section *Working with Templates* and the appropriate section for the compiler you use.

Compiler	Section
C++ compilers based on the **Cfront** translator from AT&T	*Working with Cfront-Based C++ Compilers* on page 91
SPARCompiler C++ Version 4.*x*	*Working with SPARCompiler C++* on page 108
SGI Delta/C++ compiler	*Working with the SGI Delta/C++ Compiler* on page 119
IBM AIX XLC compiler	*Working with the IBM AIX XLC C++ Compiler* on page 122
HP aC++ compiler	*Working with the HP aC++ Compiler* on page 127

Working with Templates

Many C++ compilers allow you to work with templates. A function template is a pattern for a set of functions. Similarly, a class template is a pattern for a set of classes. For example, a template for stack classes describes the form of data and functions in terms of an unspecified element type. To create a stack class, you supply a parameter, the actual type of data in the stack. Using this mechanism, you can create classes for stacks of strings or stacks of integers. These stack classes may contain different types of items, but they have the same default behaviors.

At compile or link time, the C++ compiler generates code for each template class. The code generation process is called *template instantiation*. In some cases, the process of template instantiation can conflict with **clearmake** building. These are possible symptoms of the conflict:

- **clearmake** performs unnecessary rebuilds.
- **clearmake** rebuilds an object that it could have winked in.
- The C++ compiler does not recompile a template source that changed, resulting in a link error or a run-time error.
- When **clearmake** winks in incremental repository files, information is lost.
- The output of **cleartool** commands that display configuration records (for example, **catcr** or **diffcr**) is confusing.

The exact nature of the symptoms depends on the compiler you use. For many compilers, these problems do not arise. For other compilers, you can correct these problems by modifying your makefile or the program source.

Explicit Instantiation

If your compiler supports it, we recommend that you use the ANSI C++ explicit instantiation syntax as defined in the ISO *Standard for the C++ Programming Language*. This method requires you to request instantiation explicitly in the source code. For more information about using explicit instantiation, see your C++ compiler documentation.

The Explicit Instantiation method requires more effort to use, but it allows you to control the placement of instantiated template code into object modules. Also, as part of the ANSI C++ standard, it may be the most portable solution. Using explicit instantiation does not conflict with **clearmake** build avoidance.

Alternative to Using the Procedures in This Chapter

The Cfront compilers discussed in the next section, as well as some other platform-specific compilers, use repositories of files to manage template instantiation. We recommend that you maintain repository directories and files in a VOB. The procedures in this chapter assume that you do so.

For compilers that use repositories to instantiate templates, an alternative to following the procedures in this chapter is to keep the repository outside the MVFS. However, be aware of the shortcomings of this solution:

- Changes to template source code are not detected unless your makefile includes explicit dependencies on the template source code.
- Repositories stored outside the MVFS are not available for winkin, so you must handle sharing between views manually. This process is prone to error.
- Files that use the repository may be winked in without associated template information. The missing information may cause links to fail.
- For files stored outside the MVFS, **clearmake** must rely on time stamp information for build avoidance and may not rebuild files that need to be rebuilt.
- No configuration records are maintained for files stored outside the MVFS.

Precompiled Header Files

Some compilers support options to generate precompiled header files. A precompiled header file is created as a sibling of one compilation and is referenced by later compilations. This may cause **clearmake** to track extra dependencies that can interfere with winkin. The resulting conflicts are similar to those caused by template instantiation. Check your compiler documentation to confirm that your compiler supports such options.

Working with Cfront-Based C++ Compilers

This section incorporates work contributed by ClearCase user Steve Vinoski of The Hewlett-Packard Company. Permission to use this work has been granted to Rational Software Corporation.

This section describes how to use **clearmake** to build C++ programs when you use a C++ compiler that is based on the Cfront translator from AT&T. If you use one of the following compilers, read this section:

- CenterLine C++ (with ObjectCenter)
- HP C++
- SGI OCC
- Sun SPARCompiler C++ 2.x (with Sun SPARCworks 3.x)

Cfront Template Instantiation: Interaction with clearmake

A compiler based on Cfront instantiates templates at link time. During the first step of the link phase, the compiler generates the template code the program needs. It later combines object files to create an executable.

The compiler manages template code in a repository which is, by default, a subdirectory of the build directory. In the repository, the compiler maintains a database of information to refer to during template instantiation and a cache of instantiated template source and object files. By default, the name of the repository directory is ptrepository.

By maintaining a repository, the compiler can reuse template code that earlier builds have generated. The compiler can ensure that cached template code is current with its source.

If you use **clearmake** to build C++ programs that contain template code and you are building in an MVFS directory, we recommend that you use one of the procedures in this chapter to avoid any incorrect behavior that may result. For example:

- **clearmake** rebuilds source files that use templates, regardless of whether the source changed. The result is a target that is never up to date.
- If you revert to an older version of a source file, **clearmake** invokes the linker to relink the program. However, when the linker invokes the compiler to reinstantiate template code, the compiler may fail to recompile templates that depend on the source file. The result can be a link error.
- **clearmake** rebuilds the executable rather than winking it in from another view.
- The configuration record for the final program contains confusing information about the dependency structure of the program. This problem becomes evident when you execute **cleartool** commands such as **catcr** and **diffcr**, which display configuration records.

These behaviors occur because the compiler manipulates files in the repository in a way that conflicts with **clearmake**'s build avoidance and configuration records. The result is that when a program uses templates, **clearmake** often associates incorrect configuration information with the program file and the files in the repository.

The following sections describe these issues in more detail. For information about how to resolve the incorrect behaviors, see *Models for Working with Cfront-Based Compilers* on page 95.

Link-Time Cfront Template Instantiation

This section describes how compilers that use link-time template instantiation (Cfront compilers) work.

During compilation, when the C++ compiler encounters the use of a template, it creates a *type map* file (usually named defmap). It records in this file each C++ type that it encounters and the name of the file in which the type is declared.

When the compiler links object modules into an executable, it performs a prelink step in which it examines each object module to locate unresolved template symbols. To resolve these symbols, the template instantiation system does the following:

1. Uses the information in the type map to create source files. The source files contain the template declarations, template definitions, and type declarations with which to instantiate the templates.
2. Compiles these source files in a special mode.
3. Removes from its list the template symbols that the resulting object modules resolve.
4. Adds to its list any unresolved symbols of templates to be instantiated, as found by the resulting object modules.
5. Repeats Step 1 through Step 4 until all template symbols are resolved.
6. Calls the real link editor, **ld**, to produce the executable.

This description shows that the C++ compiler does the following:

- Maintains a repository of data between executions; it saves information about C++ types in the defmap file during each compilation.
- Generates source code that is compiled into template instantiation object files.
- Performs simple link processing by searching object modules for unresolved template symbols and by attempting to resolve them.
- Performs build avoidance by detecting dependencies between template objects generated during an earlier link and both the template sources (header and definition files) and the sources that use the templates. It then reinstantiates the objects only when necessary.

The compiler performs build avoidance to minimize the costs of template instantiation. If the compiler reinstantiates the necessary templates whenever a program is linked, the use of many templates in a program requires very long link times. To minimize the overhead for subsequent builds, the compiler stores the type map and the template object modules in a directory (usually named ptrepository), so that they can be reused if possible. The compiler determines whether a template object module can be reused by analyzing the dependencies of the source files used to create it. If any of the sources have been updated since the template object module was created, the compiler must rebuild the object module.

This build-avoidance scheme is equivalent to the standard **make** scheme and simpler than the ClearCase scheme: it does not guarantee correct configurations that can be shared by multiple users in a parallel development environment.

How Link-Time Instantiation Interferes with clearmake

This section describes how typical C++ compilers that use repository-based template instantiation at link time can interfere with the incremental updating of ClearCase derived objects.

Whenever the C++ compiler encounters a template in a source file (for example, **x.cc**), it updates the defmap file. When the defmap file is created within an MVFS directory, ClearCase detects the updating activity. When **clearmake** invokes the compiler, the defmap file becomes a derived object (DO), a sibling of the object file (**.o** file). Associated with both DOs is a configuration record (CR). The CR describes the build script that was executed and the object versions (for example, of x.cc, x.h, and so on) that were accessed.

If the compilation of another source file causes the defmap file to be updated again, a new DO and CR are created for the defmap file, thereby overwriting the previous

defmap DO. Therefore, whenever the defmap file is updated, the derived objects for all targets that modified it previously are no longer candidates for winkin.

The C++ compiler updates the defmap file incrementally; that is, it does not destroy the data written there during the earlier compilation of x.cc. However, ClearCase has no knowledge of the semantics of the update. It assumes that the defmap file was completely regenerated and that its current configuration depends only on the most recent target rebuild that updated it. If another view's configuration matches that of the most recent target rebuild in the first view, clearmake can wink in the defmap file to the second view inappropriately.

Models for Working with Cfront-Based Compilers

To help **clearmake** and the C++ compiler work together effectively, we recommend that you design your makefile according to one of the models described in this chapter. The model you choose depends on your needs:

- The Simple model is the easiest to use. It works best for developers who work independently of others. It is not as effective for team projects in which sharing builds is important.
- The Multiple Repositories model is fairly easy to use and solves the problems of sharing builds in a team project. Its disadvantage is that compilations are slower.
- The Forced Instantiation model places a heavy burden on you, but it is the only model that allows you to build archives and shared libraries that contain template code.

Caution: If you use parallel builds to build applications in C++ that use templates, you must use the Multiple Repository model. This model avoids the problem of multiple processes trying to access the same template database simultaneously, which causes conflicts and build failure.

The remainder of this section describes each model in more detail.

The Simple Model

The Simple model resolves some of the conflicts between ClearCase and Cfront-based C++ compilers. It allows you to build programs, but it does not allow you to rely on **clearmake** to wink in the results of previous builds.

Insert the following line into your makefile, at any point in the file that does not break a rule or definition that spans multiple lines:

```
include ccase-home-dir/config/clearmake/cfront_simple.mk
```

Note: Substitute your ClearCase installation directory for *ccase-home-dir* (default is /usr/rational/clearcase).

How the Simple Model Works

The cfront_simple.mk makefile contains the following **clearmake** directives:

```
.SIBLINGS_AFFECT_REUSE:
.INCREMENTAL_REPOSITORY_SIBLING: ptrepository/defmap
.NO_DO_FOR_SIBLING: ptrepository/_lock    \
        ptrepository/_instfile ptrepository/defmap.old
.SIBLING_IGNORED_FOR_REUSE: ptrepository/defmap
.DEPENDENCY_IGNORED_FOR_REUSE: ptrepository/defmap.old
```

The **.INCREMENTAL_REPOSITORY_SIBLING** directive in cfront_simple.mk prevents **clearmake** from winking in object files that contain template references. This behavior is important because such files can cause the instantiation step to fail.

The **.NO_DO_FOR_SIBLING** directive tells **clearmake** to treat temporary files created by the Cfront compiler as noncritical. You can accomplish the same purpose (with less overhead) by removing these noncritical files in the build scripts that perform C++ compilations. For example, you can define a suffix rule as follows:

```
.C.o:
        $(C++C) $(C++FLAGS) -c $<
        rm -f ptrepository/_lock ptrepository/_instfile \
            ptrepository/defmap.old
```

Removing these noncritical files in the build script also reduces confusing **catcr** and **diffcr** output.

If you also build your program with other **make** programs, the directives in cfront_simple.mk are ignored (because other **make** programs do not recognize these directives).

For more information about the directives in the cfront_simple.mk makefile, see *Special Targets* on page 77.

Sample Scenario Using the Simple Model

The Simple model prevents **clearmake** from winking in an object file that contains template references to avoid an inappropriate winkin. To take advantage of the results of a previous build (for example, a nightly build), perform manual winkin by using the **cleartool winkin** command. The **–recurse** option of the **winkin** command may be used to wink in a hierarchy of files that produced a build.

For example, a build in view **nightly** produces executable hello in VOB directory /vobs/test. To wink in the hierarchy of DOs for the version of hello from that build:

1. Set your config spec to match that of **nightly**. This selects the correct version of hello. (An alternative is to specify a VOB-extended pathname for the derived object in Step 2.)

2 Type the following command:

cleartool winkin –recurse /view/nightly/vobs/test/hello

Because the **winkin –recurse** command winks in a hierarchy of DOs without regard to the makefile or config spec selections in the current view, it is a good idea to issue a **clearmake** command immediately after issuing the **winkin –recurse** command to ensure that all DOs are up to date. Invoking **clearmake** also ensures that files you have checked out are rebuilt.

For more information, see the **winkin** reference page.

Limitations of the Simple Model

The following points describe limitations of the Simple model and some workarounds.

- The compiler may reuse out-of-date template code that ought to have been reinstantiated.

  This can happen if you change a reserved checkout to unreserved or change your configuration specification in such a way that you revert to a version of a template source file that is older than the version that the compiler used in a previous build. When **clearmake** detects the change, it recompiles the target. The compiler, however, relies on a time-stamp comparison of cached template code with its source. As a result, it may inappropriately reuse code that is in the repository. (Note that regular **make** does not detect the change and does not issue a rebuild.)

  To work around the problem, you must force the compiler to reinstantiate templates, for example, by removing .o files from the repository directory.

- In certain cases, **clearmake** does not wink in object files and executables.

  The **.INCREMENTAL_REPOSITORY_SIBLING** directive in cfront_simple.mk prevents **clearmake** from winking in object files and executables that contain template code. If you want to wink in such files produced by a previous build, you must do so manually, as described in *Sample Scenario Using the Simple Model* on page 96.

- In certain cases, **clearmake** does not detect changes to template implementation source files.

  The **.DEPENDENCY_IGNORED_FOR_REUSE** directive in cfront_simple.mk avoids unnecessary rebuilding by ignoring detected dependencies on certain objects. Thus, its behavior more closely resembles that of **make**. The disadvantage of ignoring these dependencies is that after performing repeated builds, **clearmake** may not detect that a change to a template implementation source file necessitates a rebuild. To force the rebuild, you must delete the instantiation object file from the repository.

- cfront_simple.mk expects the name of the repository to be ptrepository. If you use the **–ptr** compiler option to change the name of the repository, you must edit cfront_simple.mk to reflect the change. Alternatively, edit a copy of the file and specify the copy in the include line in your makefile.

  Note: If you use **–ptr** to change the name of the directory containing the template repository but do not change the name of the repository, you do not have to edit cfront_simple.mk. The patterns in cfront_simple.mk match ptrepository in any directory.

- When you use the Simple model, the **cleartool** commands **diffcr** and **catcr** may produce unexpected output for programs that contain template code.

  The configuration record for the final program contains confusing information about the dependency structure of the program. This problem becomes evident when you execute **cleartool** commands such as **catcr** and **diffcr** that display configuration records.

The Multiple Repositories Model

The Multiple Repositories model requires you to insert lines into your makefile and to follow certain naming conventions in the build rules. **clearmake** then invokes a set of scripts that perform extra repository management to avoid conflicts with **clearmake** build avoidance. The scripts maintain a separate repository for each object file or executable that contains template code.

We recommend that you use this model if you depend on sharing builds within a development team or if the problems that the Simple model does not solve are unacceptable. The Multiple Repositories model is more difficult to use than the Simple model and requires more development time.

The following sections describe how to perform these tasks:

1. Ensure that your makefile uses a recognized C++ compiler macro.
2. Insert special build rules in your makefile.
3. Verify that the special build rules take effect.

Using a Recognized Compiler Macro

Ensure that your makefile uses one of the following macro invocations to represent the C++ compiler in every place that the compiler is called:

- **$(C++C)**
- **$(C++)**
- **$(CXX)**
- **$(CCXX)**

- **$(CCC)**

Note that **$(CC)** is not in this list because it usually designates a C compiler, rather than a C++ compiler.

The macro invocation must appear in every build rule for C++ objects. For example:

```
.C.o:
        $(C++) $(C++FLAGS) -c $<
```

The macro invocation must also appear wherever you execute the C++ linker. For example:

```
myprog: $(MYOBJS)
        $(C++) $(C++FLAGS) -o myprog $(MYOBJS)
```

Inserting Special Build Rules in Your Makefile

Edit your makefile as follows:

1 Insert the following line:

```
$(CCASE_CXX_INC_LNK)
```

You must insert this line in a specific place:

- If a C++ compiler macro is defined in your makefile, you must insert this line immediately after the line that contains the compiler macro definition, because the special build rule redefines the C++ compiler macro. However, if you are using this makefile only with **clearmake**, you can remove the line that contains the compiler macro definition.
- Otherwise, insert the new line at the top of your makefile.

With the incremental rules in effect, **clearmake** can manage the template instantiation process efficiently. In most cases, these macros work well. In some cases, you may want to use the alternate (CM-safe) multiple repository model. See *Using an Alternate (CM-Safe) Multiple Repository Model* on page 100 to determine whether you need to use alternate rules.

Note: The **CCASE_CXX_INC_LNK** macro expands to an include directive:

include *ccase-home-dir*/**config/clearmake/inc_cxx.mk**

The macro is provided so that your makefile can be used transparently with other **make** programs. If that is not important to you, you can use the include directive instead to enable the special build rules.

2 Decide whether to define the pathname to the C++ compiler, as follows:

a If the name of the compiler is **CC** and it does not require preceding pathname components, do nothing.

b Otherwise, insert a line in the format

`REAL_C++=`*your-CC-program*

where *your-CC-program* is the name of the compiler as it appears in the definition of the C++ compiler macro.

If you use Rational Purify with this model, insert the path to Purify before the path to the C++ compiler. For example:

REAL_C++='purify CC'

Note that you must use single quotes.

Similarly, if you use PureLink with this model, insert the path to PureLink before the path to the C++ linker, for example:

```
ATRIA_C++LINK=${CCASE_MAKE_CONFIG_DIR}/atria_cxxl \
     $@ cfront timestamp_cache 'purelink $(REAL_C++)'
```

If you are using the CM-safe model discussed in the next section, substitute **CM-safe** for **timestamp-cache**:

```
ATRIA_C++LINK=${CCASE_MAKE_CONFIG_DIR}/atria_cxxl \
     $@ cfront CM-safe 'purelink $(REAL_C++)'
```

Using an Alternate (CM-Safe) Multiple Repository Model

In most cases, the incremental build rules described in the previous section work well. However, if unused template code accumulates in the repository, the build rules may prevent winkin.

To use the CM-safe build rules, substitute the following line of code for the link-time template instantiation build rule:

```
$(CCASE_CXX_SAFE_LNK)
```

With the CM-safe rules in effect, **clearmake** forces the C++ compiler to instantiate all templates every time it links the program. The result is that there are no superfluous dependencies to prevent **clearmake** from winking in object files. The disadvantage of using CM-safe build rules is that for every link, extra time is spent on template instantiation.

Note: The **CCASE_CXX_SAFE_LNK** macro expands to an include directive:

include *ccase-home-dir*/**config/clearmake/cmsafe_cxx.mk**

The macro is provided so that your makefile can be used transparently with other **make** programs. If that is not important to you, you can use the include directive instead to enable the special build rules.

Example Makefile Using the Multiple Repository Model

Consider the following makefile:

```
MYOBJS=main.o std.o istring.o site.o point.o

.SUFFIXES: .C .o
.C.o:
        $(C++) $(C++FLAGS) -c $<

C++=/net/elm/tools/bin/CC
C++FLAGS=-g

myprog: $(MYOBJS)
        $(C++) $(C++FLAGS) -o myprog $(MYOBJS)
```

After you make the modifications described in *Inserting Special Build Rules in Your Makefile* on page 99, the makefile looks like this:

```
MYOBJS=main.o std.o istring.o site.o point.o

.SUFFIXES: .C .o
.C.o:
        $(C++) $(C++FLAGS) -c $<

C++=/net/elm/tools/bin/CC  (this line can be removed if the makefile is being used only with clearmake)
C++FLAGS=-g
$(CCASE_CXX_INC_LNK)
REAL_C++=/net/elm/tools/bin/CC

myprog: $(MYOBJS)
        $(C++) $(C++FLAGS) -o myprog $(MYOBJS)
```

Testing the Makefile

To test your modifications:

1. Verify that your makefile is still valid for any **make** programs other than **clearmake** that you use to build your program. Your modifications must not affect how other **make** programs build your program.

2 When you build your program with **clearmake**, check whether there is a difference in the commands that **clearmake** executes:

- If your build rules instantiate the template at link time, **clearmake** executes a C++ translation whenever it echoes a command line that begins with the following text:

  ccase-home-dir`/config/clearmake/atria_cxx cfront timestamp-cache`

- If you used CM-safe build rules, **clearmake** echoes a command line that begins with the following text whenever it executes a C++ translation:

  ccase-home-dir`/config/clearmake/atria_cxx cfront cm-safe`

If the result of either of these tests is not what you expect, examine your makefile to verify that you modified it correctly.

How the Multiple Repositories Model Works

When **clearmake** executes the build rule, the C++ macro expands to code that includes the file *ccase-home-dir*/config/clearmake/inc_cxx.mk. If you use the CM-safe rules, **clearmake** includes the cmsafe_cxx.mk file instead. These files add special build rules for the Cfront compilers. Because the macros are ordinarily undefined, other **make** programs expand the macro lines to empty text, which is ignored.

The inc_cxx.mk and cmsafe_cxx.mk files provide a wrapper around the compiler and linker. The wrapper runs the **atria_cxx** script whenever the compiler or linker is invoked. The script performs actions that work with the compiler's template instantiation method and also runs other scripts. To cause the **atria_cxx** script to echo the command lines it executes, invoke **clearmake** with the **–v** or **–d** option or set the CCASE_VERBOSITY environment variable as described on the **env_ccase** reference page.

By default, the C++ compiler creates only one directory, ptrepository, for the template instantiation repository. The effect of modifying the build rules is as follows:

- The compiler creates an *xxx*.ptrep directory for each C++ target, regardless of whether the target is an object file or a linked executable. Maintaining separate repositories is an important part of the Multiple Repositories solution to the conflict between **clearmake** and Cfront template instantiation.
- **clearmake** runs the **atria_make_ptrep** script to combine the repositories before the link.

- The incremental macro calls the **atria_make_ts_cache** script to create a time-stamp cache. It uses this cache to determine when rebuilds are necessary.

  The CM-safe rules do not use a time-stamp cache because they always delete and re-create the entire repository.

As a result of these actions, the new build rules facilitate better sharing (winkin) by avoiding incremental updates to repository files. The CM-safe rules also prevent the use of existing template information, which may create superfluous dependencies.

Limitations of the Multiple Repositories Model

The Multiple Repositories model has the following limitations:

- You cannot use the **–ptr** compiler option.

  To maintain separate template repositories, the build script inserts a **–ptr** option in each C++ command line. Because the build rules assume control over the repository directories, you cannot use the **–ptr** option to specify another repository directory.

- An error during instantiation invalidates all template code.

  The compilers discussed in this section instantiate templates at link time. If an error occurs during template instantiation, it is probably a compilation error in a template source file. When such an error occurs, the link fails and **clearmake** aborts the build.

  When the build is interrupted in this way, the **clearmake** scripts do not record configuration information for the template code in the repository. On the next build, the scripts note the absence of configuration information and cause the C++ compiler to reinstantiate all the template code required by the program. This limitation can slow the development process.

- In certain cases, **clearmake** does not wink in repository files.

  The template repository for the final executable accumulates files as the build progresses. The Cfront compiler and linker do not delete files from the repository. As a result, a repository may contain template code that is no longer used by the program. Normally, the contents of repositories can be shared (winked in) among views if the program and its dependencies match across the views. However, the presence of unused template code in a repository can interfere with sharing. To remove unused template code accumulated in a repository, delete the entire repository directory and rebuild.

- Multiple repositories can hold duplicates of instantiation objects.

Because a separate template repository is maintained for each object file and executable, the compiler is prevented from reusing template code that was instantiated during the compilation of another module, even if the instantiations are identical. Thus, multiple repositories can consume more disk space than a single repository. For an example of how to eliminate duplicate instantiation objects from an archive library, see Step 3 on page 106.

The Forced Instantiation Model

Compilers that are based on Cfront instantiate templates only during the link step of a program build. These compilers determine which templates to instantiate by examining external references in the object files to be linked; the programmer does not have direct control over which templates to instantiate. When you need to control template instantiation, use the Forced Instantiation model.

The basic strategy of the Forced Instantiation model is to prevent the linker from instantiating templates automatically. To do so, you instantiate all templates that are needed by an executable or by an archive library, and then name them explicitly for inclusion in that executable at link time.

The Forced Instantiation model places the burden of managing template instantiation on you. However, it is the appropriate model to use in situations that require control over instantiation. When you use this model exactly as presented here, no conflict arises between the C++ compiler and **clearmake** building.

Use the Forced Instantiation model in the following situations:

- If you are developing code for multiple platforms, some compilers may not provide an automatic instantiation mechanism and will require a model similar to this one for instantiating templates. Using a common method of instantiation across multiple platforms may be easiest for you.
- If you build an archive or shared library containing instantiated template code, you must force the compiler to instantiate the code you need and extract the resulting object files from the repository.

To use this model:

- Maintain a set of “dummy” source files.
- Set up your makefile to build a dummy executable and build template code into the final executable or library.

Maintaining Dummy Source Files

The Forced Instantiation model requires you to maintain a set of dummy source files. Each dummy source file is a C++ source file that contains references to a set of template classes or functions. For each template class that you want to instantiate, add one variable declaration to the file. For each template function, add a call to that function.

Your project’s needs determine how to distribute the references among the dummy files. You may want to have one dummy file that contains all references, or you may want to group the references to reflect how the sources in your program are divided into libraries and executables.

For example, if **Array<char>** and **List<String>** are template classes your program requires, you can produce the following dummy file:

```
// dummy source file for forcing template instantiation.

#include "array.h"      // declares class template Array
#include "list.h"       // declares class template List
#include "string.h"     // declares class String

Array<char> dummy0;     // assumes void constructor
List<String> dummy1;    // assumes void constructor
```

Additional instructions for working with dummy source files:

- Include in the dummy source file the header files that contain requisite definitions.
- Define variables, as needed, to provide as arguments to constructors.
- If the template class you need to instantiate has no public constructors, you may substitute a line for the variable declaration of the form

  `((`*Class* `*)0)->`*MemberFunction* `();`

  where *Class* is the name of the class and *MemberFunction* is a public member function of that class.
- If the template class has no public member functions, it must declare a friend class or friend function to be usable. In this case, write a dummy version of a friend function (or a member function of a friend class) in the file and put the variable declaration into the dummy friend function. Because this code is not linked into your program, you do not need to worry about multiple symbol definitions.

Setting Up the Makefile

To set up your makefile:

1 Assign separate repositories to each object file and executable.

Assign each object file and executable its own repository by specifying the **–ptr** compiler option on every C++ compiler command line. Add the following to the options you pass to the C++ compiler:

```
-ptr$@.ptrep
```

For example, if **$(C++FLAGS)** is expanded on every C++ compiler command line, add the option to the macro definition.

```
C++FLAGS=-ptr$@.ptrep    ...other compiler options...
```

2 Instantiate the templates by building the dummy executables.

The following example shows a **make** rule that compiles the dummy.C source file into the dummy.out executable, producing the dummy.out.ptrep repository. The **–pta** compiler option is required. This option directs the compiler to instantiate all template functions in a class rather than instantiating only the template functions that are referenced.

```
dummy.out: dummy.C main.o std.o
        rm -rf $@.ptrep
        $(C++) $(C++FLAGS) -pta $@ dummy.C main.o std.o
```

This **make** rule deletes the existing repository before rebuilding. The rule prevents reuse of existing template code, but ensures clean configuration information for the newly instantiated template code. It is important that you follow this model in your makefile, especially to ensure that the objects you build can be shared (winked in) by other views.

The example rule in this section depends on the additional object files, main.o and std.o. Add other objects to your makefile rule as necessary to link the program. In this section's example, assume that main.C contains an empty **main( )** function, and std.C contains other definitions required to link the program.

3 Build the final executable or library.

a Make the final executable or library target depend on the dummy executables.

b In the build rules for the final target, invoke the **atria_list_obj** utility to collect all the names of the template object files produced by the compilations of the dummy sources.

c Insert the names of the object files collected in Step b onto the compiler or **ar** command line.

For example, suppose the target is the archive lib.a. The dummy source files dummy0.C, dummy1.C, and dummy2.C contain template references required by lib.a. Suppose also that **$(OBJECTS)** expands to a list of the other objects required by lib.a.

In the following example, the utility program, **atria_list_obj** takes one or more repository directories as command line arguments and prints a list of the .o files that it finds in those directories or any of their subdirectories. This example uses **atria_list_obj** to collect the names of template object files and to put them on the **ar** command line.

```
lib.a: $(OBJECTS) dummy0.out dummy1.out dummy2.out
        rm -f $@
        ar qcv $@ $(OBJECTS) \
        `atria_list_obj dummy0.out.ptrep \
            dummy1.out.ptrep dummy2.out.ptrep`
```

For more information, examine the code for the utility program in *ccase-home-dir*/config/clearmake/atria_list_obj.

How the Forced Instantiation Model Works

By default, the C++ compiler creates only one directory, ptrepository, for the template instantiation repository. When you modify the build rules, the compiler creates an *xxx*.ptrep directory for each C++ target, regardless of whether the target is an object file or a linked executable. Maintaining separate repositories is an important part of the Forced Instantiation solution to the conflict between ClearCase and Cfront template instantiation.

When you specify the **–pta** option to the linker, the linker forces the instantiation of all template objects when it builds the dummy output executable. The template instantiation objects, which behave like a dictionary, are then available for placement on a command line to build a library or final executable. The **atria_list_obj** utility aids this process.

As a result of these actions, the new build rules facilitate better sharing (winkin) by avoiding incremental updates to repository files.

Limitations of the Forced Instantiation Model

- The dummy source files must be kept in sync with the actual uses of templates in the production library or executables.
- You must use the **–ptr** compiler option as described in Step 1 on page 106. You cannot use a single repository.

- Multiple repositories can hold duplicates of instantiation objects.

  Because a separate template repository is maintained for each object file and executable, the compiler is prevented from reusing template code that was instantiated during the compilation of another module, even if the instantiations are identical. Thus, multiple repositories can consume more disk space than a single repository does. For an example of how to eliminate duplicate instantiation objects from an archive library, see Step 3 on page 106.

Working with SPARCompiler C++

This section describes how to use **clearmake** to build C++ programs using SPARCompiler C++ Version 4.*x*. This compiler is bundled with SPARCworks Version 3.0 or later. If you use this compiler, read this section. If you use an earlier version of SPARCompiler C++, see *Working with Cfront-Based C++ Compilers* on page 91.

Note: The **clearmake** models outlined here are accurate as of the time that this document was written, but may become obsolete or incomplete as new versions of the compiler are released.

SPARCompiler Template Instantiation: Interaction with clearmake

When the SPARCompiler C++ compiles a source module into an object module, it instantiates all the template code to which the module refers. The compiler places the generated template code into a repository directory, Templates.DB. By default, the repository is a subdirectory of the build directory. At link time, the compiler links instantiated template code from the repository into the final executable as needed to resolve references in the program. The instantiated template code remains in the repository, so that the compiler can reuse template code generated by earlier builds. The compiler employs its own scheme to check that cached template code is up to date with its source.

If you use **clearmake** to build C++ programs that contain template code and you are building in an MVFS directory, we recommend that you use one of the procedures in this section to avoid any incorrect behavior that may result. For example:

- **clearmake** rebuilds components of the program, regardless of whether the source changed. The result is a target that is never up to date.
- If you revert to an older version of a source file, **clearmake** invokes the compiler to recompile the source files that depend on the template. However, the C++ compiler may fail to reinstantiate the template code.

- **clearmake** rebuilds program components rather than winking them in from another view.
- The configuration record for the program components contains confusing information about the dependency structure of the program. This problem becomes evident when you execute **cleartool** commands such as **catcr** and **diffcr**, which display configuration records.

These behaviors occur because the compiler manipulates the files in the repository directory in a way that conflicts with **clearmake**'s build avoidance and configuration records. When a program uses templates, **clearmake** often associates incorrect configuration information with the program file and the files in the repository.

Setting Up the Repository

By default, the repository directory, Templates.DB, is a subdirectory of the compiler's working directory. (Although you can specify the **–ptr** compiler option to change the repository directory, we recommend that you use the default directory because the makefile models in this section assume that you are using the default directory.) For each repository directory in your build system, follow these steps to set up the repository:

1. Verify that the Templates.DB subdirectory is a ClearCase directory element (under version control).
2. Verify that the file Templates.DB/Template.opt exists and is a ClearCase element. If it does not exist, create an empty file and convert it to an element.

   The SPARCompiler input file Template.opt tracks template specialization, that is, a specialized implementation of a template class member or of a template function. You must always keep the Template.opt file under version control, even if you do not use template specializations, to avoid conflicts with the **clearmake** build avoidance mechanism.

   Note: If you are using version 4.1 or later of the SPARCompiler, it is not necessary to make Template.opt a ClearCase element. However, we recommend that you do so if you are adding content to this file.

Cleaning the Repository

To force a complete rebuild or to reclaim disk space, you may want to clean the repository. A common practice is to add a target to your makefile to delete the entire Templates.DB repository. However, because we recommend that you maintain Templates.DB as a versioned directory, you may not delete the entire directory. To remove the derived objects from the repository while leaving the versioned files, remove only the files that match the following wildcard patterns:

```
Templates.DB/*.o
Templates.DB/*.state
Templates.DB/*.system
Templates.DB/*.module
Templates.DB/Module.DB/*.module
```

Models for Working with SPARCompiler C++

To help **clearmake** and the SPARCompiler C++ compiler work together effectively, we recommend that you design your makefile according to one of the models described in this section. The model you choose depends on your needs:

- The Simple model is easy to use. It works best for developers who work independently. It is not as effective for team projects in which sharing of builds is important.
- The Multiple Repositories model is fairly easy to use and solves the problems of sharing builds in a team project. Its disadvantage is that compilations are slower.

There is no Forced Instantiation model for this compiler, because this compiler does not provide a way to disable automatic template instantiations.

The remainder of this section describes each model.

The Simple Model

The Simple model solves some of the conflicts between Rational ClearCase and SPARCompiler C++ compilers. It allows you to build programs.

Insert the following line into your makefile, at any point in the file that does not break a rule or definition that spans multiple lines.

include *ccase-home-dir*/config/clearmake/sunpro_4_0_simple.mk

How the Simple Model Works

The sunpro_4_0_simple.mk makefile contains the following **clearmake** directives.

```
.SIBLINGS_AFFECT_REUSE:
.SIBLING_IGNORED_FOR_REUSE: Templates.DB/%.system \
        Templates.DB/%.state Templates.DB/%.module
.DEPENDENCY_IGNORED_FOR_REUSE: Templates.DB/Dependency.state \
        Templates.DB/%.module Templates.DB/Module.DB/%.module
```

If you also build your program with other **make** programs, the directives in sunpro_4_0_simple.mk are ignored (because other **make** programs do not recognize these directives).

For more information about each of these directives, see *Special Targets* on page 77.

Sample Scenario Using the Simple Model

To avoid an inappropriate winkin, the Simple model prevents **clearmake** from winking in an object file that contains template references. To take advantage of the results of a previous build (for example, a nightly build), perform manual winkin by using the **cleartool winkin** command. The **–recurse** option of the **winkin** command may be used to wink in a hierarchy of files that produced a build.

For example, a build in view **nightly** produces executable hello in VOB **/vobs/test**. To wink in the hierarchy of DOs for the version of hello from that build:

1. To select the correct version of hello, set your configuration spec to match that of **nightly**. (An alternative is to specify a VOB-extended pathname for the derived object in Step 2.)
2. Type the following command:

   cleartool winkin –recurse /view/nightly/vobs/test/hello

Because the **winkin –recurse** command winks in a hierarchy of DOs without regard to the makefile or config spec selections in the current view, it is a good idea to issue a **clearmake** command immediately after issuing the **winkin –recurse** command to ensure that all DOs are up to date. Invoking **clearmake** also ensures that files you have checked out are rebuilt.

For more information, see the **winkin** reference page.

Limitations of the Simple Model

These are the limitations of the Simple model and the workarounds for each:

- The compiler may reuse template code that ought to have been reinstantiated.

  This can happen if you change a reserved checkout to unreserved or change your configuration specification in such a way that you revert to a version of a template source file that is older than the version that the compiler used in a previous build. When **clearmake** detects the change, it recompiles the program template. The compiler, however, relies on a file time-stamp comparison to check cached template code against its source. As a result, it may inappropriately reuse code that is in the repository.

  To work around the problem, you must force the compiler to reinstantiate templates, for example, by removing the .o files from the repository directory.

- In certain cases, **clearmake** does not detect changes to template implementation source files.

The **.DEPENDENCY_IGNORED_FOR_REUSE** directive in sunpro_4_0_simple.mk avoids unnecessary rebuilding by ignoring detected dependencies on certain objects. Thus, its behavior more closely resembles that of **make**. The disadvantage of ignoring these dependencies is that after performing repeated builds, **clearmake** may not detect that a change to a template implementation source file necessitates a rebuild. To force the rebuild, you must delete the instantiation object file from the repository.

- sunpro_4_0_simple.mk expects the name of the repository to be Templates.DB. If you use the **–ptr** compiler option to change the name of the repository, you must edit sunpro_4_0_simple.mk to reflect the change. Alternatively, edit a copy of the file and specify the copy in the include line in your makefile.

  Note: If you use **–ptr** to change the name of the directory containing the template repository but do not change the name of the repository, you do not have to edit sunpro_4_0_simple.mk. (The patterns in sunpro_4_0_simple.mk match Templates.DB in any directory.)

- The configuration record for the program components contains confusing information about the dependency structure of the program.

  The **cleartool** commands **catcr** and **diffcr** often produce unexpected output for programs that contain template code.

Building Archives That Contain Template Code

If you use SPARCompiler C++ compiler with the Simple model, building an archive that contains instantiated template code is relatively easy. The next few sections describe how to perform the following steps to set up the makefile:

1. Identify a set of source files that refers to all the template code you require.
2. Compile the source files that you identified in Step 1.
3. Make the archive target depend on the compilation of the source files.
4. Pass the names of the object files in the repository to the archiver.

Managing Template References

The following points describe how to manage template references, depending on which symbols you want to define in your archive:

- You want to define in the archive every template symbol referenced by the files in your archive.

  You may use the default repository Templates.DB.

- There are template symbols referenced by the files in the archive that you do not want to be defined in the archive.

  You must specify separate repositories for the templates to be archived and for the templates that are not to be archived, and build your sources accordingly. You can use the **–ptr** compiler option to direct the compiler to look for a repository in a directory other than the current working directory.

- There are template symbols you want to define in your archive, but they are not referenced in any of the object files in the archive.

  You must force the compiler to instantiate the templates you need. Create one or more dummy source files that refer to the missing symbols.

Building the Archive

In the following example, the target is the archive, lib.a, and **$(OBJECTS)** expands to a list of objects required by lib.a.

The source file template_refs.C refers to additional template code that you want to include in the archive. The template code to be archived resides in the default repository, ./Templates.DB.

In the following example, the utility program, **atria_list_obj**, takes one or more repository directories as command line arguments and prints a list of the .o files that it finds in those directories or any of their subdirectories. The example uses **atria_list_obj** to collect the names of template object files and to put them on the **ar** command line.

```
lib.a: $(OBJECTS) template_refs.o
        rm -f $@
        ar qcv $@ $(OBJECTS)  \
            `atria_list_obj Templates.DB`
```

For more information, examine the code for the utility program in *ccase-home-dir*/config/clearmake/atria_list_obj.

The Multiple Repositories Model

The Multiple Repositories model requires you to insert lines into your makefile and to follow certain naming conventions in the build rules. **clearmake** then invokes a set of scripts that perform extra repository management to avoid conflicts with **clearmake** build avoidance. The scripts maintain a separate repository for each object file or executable that contains template code.

This model provides the benefit of winking in the results of previous builds. However, it prevents the compiler from reusing template code cached previously, so compiling takes longer. If you build archives containing template code, you may find the Multiple Repositories model too complicated.

The following sections describe how to perform these tasks:

1. Ensure that your makefile uses a recognized C++ compiler macro.
2. Insert special build rules in your makefile.
3. Verify that the special build rules take effect.

Using a Recognized Compiler Macro

Ensure that your makefile uses one of the following macro invocations to represent the C++ compiler in every place that the compiler is called:

- **$(C++C)**
- **$(C++)**
- **$(CXX)**
- **$(CCXX)**
- **$(CCC)**

Note that **$(CC)** is not in this list because it usually designates a C compiler, rather than a C++ compiler.

The macro invocation must appear in build rules for C++ objects. For example:

```
.C.o:
      $(C++) $(C++FLAGS) -c $<
```

The macro invocation must also appear wherever you execute the C++ linker. For example:

```
myprog: $(MYOBJS)
      $(C++) $(C++FLAGS) -o myprog $(MYOBJS)
```

Inserting Special Build Rules in Your Makefile

Edit your makefile as follows:

1. Insert the following line:

   ```
   $(CCASE_CXX_COMP)
   ```

   - If a C++ compiler macro is defined in your makefile, you must insert this line immediately after the line that contains the compiler macro definition, because the special build rule redefines the C++ compiler macro. However, if you are using this makefile only with **clearmake**, you can remove the line containing the compiler macro definition.
   - Otherwise, insert the new line at the top of your makefile.

   With these rules in effect, **clearmake** can manage the template instantiation process efficiently.

Note: The **CCASE_CXX_COMP** macro expands to an include directive:

include *ccase-home-dir*/**config/clearmake/comp_cxx.mk**

The macro is provided so that your makefile can be used transparently with other **make** programs. If that is not important to you, you can use the include directive instead to enable the special build rules.

2 Decide whether to define the pathname to your C++ compiler, as follows:

a If the name of your compiler is **CC** and does not require preceding pathname components, do nothing.

b Otherwise, insert a line in the format

```
REAL_C++=your-CC-program
```

where *your-CC-program* is the name of the compiler as it appears in the definition of the C++ compiler macro.

If you use Rational Purify with this model, insert the path to Purify before the path to the C++ compiler. For example:

```
REAL_C++='purify CC'
```

Similarly, if you use PureLink with this model, insert the path to PureLink before the path to the C++ linker. For example:

```
ATRIA_C++LINK=${CCASE_MAKE_CONFIG_DIR}/atria_cxxl \
        $@ SunCC SunCC4.0 'purelink $(REAL_C++)'
```

Example Makefile Using the Multiple Repositories Model

Consider the following makefile:

```
MYOBJS=main.o std.o istring.o site.o point.o

.SUFFIXES: .C .o

.C.o:
      $(C++) $(C++FLAGS) -c $<

C++=/opt/SUNWspro/bin/CC
C++FLAGS=-g

myprog: $(MYOBJS)
      $(C++) $(C++FLAGS) -o myprog $(MYOBJS)
```

After you make the modifications described in *Inserting Special Build Rules in Your Makefile* on page 114, the makefile looks like this:

```
MYOBJS=main.o std.o istring.o site.o point.o

.SUFFIXES: .C .o
```

```
.C.o:
      $(C++) $(C++FLAGS) -c $<

C++=/opt/SUNWspro/bin/CC  (line can be removed if makefile is only being used with clearmake)
C++FLAGS=-g
$(CCASE_CXX_COMP)
REAL_C++=/opt/SUNWspro/bin/CC

myprog: $(MYOBJS)
      $(C++) $(C++FLAGS) -o myprog $(MYOBJS)
```

Testing the Makefile

To test your modifications:

1. Verify that your makefile is still valid for any **make** programs other than **clearmake** that you use. Your modifications must not affect the way **make** programs other than **clearmake** build your program.
2. When you build your program with **clearmake**, check whether there is a difference in the commands that **clearmake** executes. Whenever **clearmake** executes a C++ translation, it echoes a command line that begins with the following text:

   ccase-home-dir`/config/clearmake/atria_cxx SunCC SunCC4.0`

If the result of either of these test steps is not what you expect, examine your makefile to verify that you modified it correctly.

How the Multiple Repositories Model Works

When **clearmake** executes the build rule, the C++ macro expands to code that includes the file *ccase-home-dir*/config/comp_cxx.mk. This file adds the special build rules suited to the Sun 4.*x* compilers. Because the macro is ordinarily undefined, other **make** programs expand the macro lines to empty text, which is ignored.

The comp_cxx.mk file provides a wrapper around the compiler and linker. The wrapper runs the **atria_cxx** script whenever the compiler or linker is invoked. The script performs actions that work with the compiler's template instantiation method, and also runs other scripts. To cause the **atria_cxx** script to echo the command lines it executes, invoke **clearmake** with the **–v** or **–d** command line option, or set the CCASE_VERBOSITY environment variable as described in the **env_ccase** reference page.

By default, the C++ compiler creates only one directory, Templates.DB, for the template instantiation repository. When you modify the build rules, the compiler creates an *xxx*.ptrep directory for each C++ target, regardless of whether the target is an object file or a linked executable. The link process uses multiple **–ptr** options to search the repositories. The new build rules eliminate unnecessary rebuilds by preventing multiple targets from writing to the same file. Therefore, maintaining separate

repositories is an important part of the solution to the conflict between **clearmake** building and Sun compiler template instantiation.

Limitations of the Multiple Repositories Model

These are the limitations of using the macro described in this section:

- You cannot use the **–ptr** compiler option.

  To maintain separate template repositories, the build script inserts a **–ptr** option onto each C++ command line. Because the build rules assume control over the repository directories, you cannot use the **–ptr** option to specify another repository directory.

- The compiler does not reuse template code.

  The build script for invocations of this compiler re-creates the template repository for the target object on every build. The rebuilds prevent the compiler from reusing template code that was instantiated during a previous build of the same module. They also prevent the compiler from sharing template code between modules. Therefore, compiling some files will take longer when you use **clearmake**.

  In addition, multiple repositories may consume more disk space than a single repository does. For an example of how to eliminate duplicate instantiation objects from an archive library or executable, see *Multiple Repositories Example* on page 118.

Building Archives That Contain Template Code

This section describes how to use the Multiple Repositories model to build an archive or shared library that contains instantiated template code. Use the Multiple Repositories model for this purpose only when necessary. We suggest that where possible, you use the Simple model to build archives or shared libraries that contain instantiated template code.

To build a library that contains instantiated template code, you must pass to **ar** or to the linker the names of all repository files that contain the code you want to include in the library or executable. In the Multiple Repositories model, the files you need may reside in different repositories. Also, duplicate files may exist among repositories.

The makefile rules that build the library must specify all directories that contain template code. The rules must also eliminate duplicate code among the repositories.

To modify makefile rules to build a library:

1 Identify the source files that refer to all the template code you require.

- Usually, you want to include in the library all template code referenced by the library source code. The set of source files that refers to all the template code you require (the reference set) is the set of library source files.
- To exclude code, identify the source files that refer to it and remove them from the reference set.
- To include in the archive template symbols that are not referenced by any of the source files in the reference set, you must force the compiler to instantiate these templates. Create one or more dummy source files that refer to the missing symbols. Add the dummy sources to the set of sources to compile.

The source files that you have identified will fall into the following categories:

- Library source files that are members of the reference set
- Library source files that are not members of the reference set
- Dummy reference source files

The second and third categories are usually empty.

2 Build the library.

a Make the archive target depend on the compilation of the source file list.

b Collect the instantiated template code into a single directory.

c Pass the names of the object files in the directory to **ar** or to the linker.

The makefile rule that builds your library must depend on the compilation of all library sources and all dummy reference sources. When the rule invokes the linker or archiver, it must pass the names of all these files:

- Library object files
- Template object files derived from library source files that are members of the reference set
- Template object files derived from dummy reference source files

Multiple Repositories Example

In the following example:

- The target is an archive, lib.a.
- **$(OBJECTS)** expands to a list of objects required by lib.a.
- **$(OBJECTS_EXCL_TMPL)** expands to a list of objects required by lib.a but containing references to templates you want to exclude from the library.

- **$(DUMMYOBJ)** expands to a list of objects built from dummy reference source files.

The repositories are named *basename*.o.ptrep, where *basename*.o is the name of a compiled C++ object module. The makefile rule creates a temporary repository called lib.a.ptrep to collect the template object files before archiving them.

The utility program, **atria_list_obj**, takes one or more repository directories as command line arguments and prints a list of the .o files that it finds in these directories or any of their subdirectories. The example uses this program to collect the names of template object files and to put them on the **ar** command line.

```
lib.a: $(OBJECTS) $(OBJECTS_EXCL_TMPL) $(DUMMYOBJ)
       rm -r -f $@ $@.ptrep
       mkdir $@.ptrep
       for i in $(OBJECTS) $(DUMMYOBJ) ; do \
          cp /dev/null `atria_list_obj $$i.ptrep` $@.ptrep ; \
       done
       ar qcv $@ $(OBJECTS) $(OBJECTS_EXCL_TMPL) \
          `atria_list_obj $@.ptrep`
       ranlib lib.a
       rm -r $@.ptrep
```

This build script copies all the required template code into a single directory, and then archives all the code in the directory. The extra copy step eliminates duplicate template code from the archive.

For more information, examine the code for the utility program in *ccase-home-dir*/config/clearmake/atria_list_obj.

Working with the SGI Delta/C++ Compiler

This section describes how to use **clearmake** effectively to build C++ programs using the SGI Delta/C++ compiler. This compiler is available with SGI C++ Version 4.0 and later.

You may be using a version of the compiler that is earlier than Version 4.0. Both the Delta/C++ compiler and the earlier compiler, which is based on the Cfront translator from AT&T, are packaged with SGI C++ Version 4.0 and later.

You are using the earlier compiler if one of the following is true:

- You are using a version of SGI C++ that is earlier than Version 4.0.
- You are using Version 4.0 of SGI C++, and you run the **OCC** executable to invoke the compiler.
- You are using Version 4.0 of SGI C++, and you specify the **–use_cfront** command-line option to the compiler.

If you are using the earlier compiler, read *Working with Cfront-Based C++ Compilers* on page 91 instead of this section.

SGI Delta/C++ Compiler Template Instantiation: Interaction with clearmake

The Delta/C++ compiler offers several methods for building template code; the method you choose depends on your needs. The following sections describe these methods.

Automatic Instantiation

By default, the SGI Delta/C++ compiler instantiates templates automatically to build template code. During a prelink step, the compiler determines what template code the program requires and compiles the necessary code into some of the object files that make up the program. The compiler tracks how the program uses template code. It records this information in the ii_files subdirectory of the build directory.

If you use the Automatic Instantiation method, **clearmake** sometimes executes unnecessary rebuilds of program components. When the prelinker compiles template code into an existing object file, the dependency information that **clearmake** previously recorded for that object file is no longer up to date. The next time that **clearmake** is invoked, it rebuilds the object file. After this rebuild, the dependency information for the object file is again correct. At this point, **clearmake** no longer rebuilds that object file unnecessarily.

The Automatic Instantiation method is the easiest to use because it requires no programmer intervention. It is suitable for most applications. Aside from the unnecessary rebuilds described above, this method does not conflict with ClearCase configuration management.

Compile-Time Demand Instantiation

The Compile-Time Demand Instantiation method instantiates templates at compile time, rather than during a prelink step. To use this method, specify the **–ptused** and **–no_prelink** compiler options to the Delta/C++ compiler.

These options cause the compiler to compile all the template code that the source module refers to into the object module. If multiple source modules refer to the same template class or function, copies of the compiled template code appear in multiple object modules. When the program is linked, the linker removes duplicate template code originating from multiple object modules.

The Compile-Time Demand Instantiation method is easy to use, requiring only that you specify extra compiler options. It is suitable for most applications, especially for building archives and shared libraries. Also, this method does not conflict with ClearCase configuration management. The disadvantage is that the compiler requires extra time and disk space to perform redundant template instantiation.

Explicit Instantiation

The Explicit Instantiation method is an alternative form of compile-time instantiation. The Delta/C++ compiler allows you to add directives to the source code to specify which template classes to instantiate.

When compiling a source module, the compiler instantiates all the template classes specified by the directives in the source. The compiler instantiates each template classes completely; that is, it instantiates every member function and static data member of the class.

To use the Explicit Instantiation method:

1. For each template class to instantiate, add one **#pragma instantiate** directive to the source code. For example, if the program requires the **Array<String>** class, add the following directive:

   ```
   #pragma instantiate Array<String>
   ```

2. In each source file that contains a **#pragma instantiate** directive, include the header files that contain definitions of the templates and classes used in the directives.

3. [Optional] Disable automatic instantiation by specifying the **–ptnone** and **–no_prelink** compiler options. Automatic instantiation does not interfere with explicit instantiation, but you can disable it, if you prefer.

The Explicit Instantiation method requires more effort to use. However, it allows you to control the placement of instantiated template code into object modules. This control is useful in some situations, especially when building archives of instantiated template code. Using explicit instantiation does not conflict with **clearmake** build avoidance.

Working with the IBM AIX XLC C++ Compiler

This section describes how to use **clearmake** effectively to build C++ programs using the IBM AIX XLC compiler.

XLC Compiler Template Instantiation: Interaction with clearmake

The XLC compiler instantiates a template when it encounters a **#pragma** directive in the source code. You can insert these directives into the source code manually, or you can direct the compiler to generate auxiliary source code that contains the necessary directives.

The XLC compiler's automatic template instantiation works as follows:

1. As the XLC compiler compiles a source module into an object module, it notes the templates that the module refers to.
2. When the compiler finishes the translation, it generates auxiliary source modules in a repository directory. By default, the repository directory is tempinc**.**
3. At link time, the compiler compiles the auxiliary sources and links the resulting object modules into the final executable. The template source and object modules remain in the repository, so that the compiler can reuse template code generated by earlier builds. The compiler verifies that cached template object modules are up to date with their sources.

If you use **clearmake** to build C++ programs that contain template code and are building in an MVFS directory, we recommend that you follow one of the procedures in this section to avoid any incorrect behavior that may result. For example:

- **clearmake** executes unnecessary recompiles of program components.
- **clearmake** fails to wink in available derived objects.
- After a template source file changes, **clearmake** does not rebuild.
- **clearmake** rebuilds the executable, but the compiler fails to reinstantiate template code for which the source has changed.
- The output of **cleartool** commands such as **catcr** and **diffcr** that display configuration records is confusing.

This behavior occurs because the compiler manipulates files in the repository in a way that conflicts with **clearmake** build avoidance and configuration records. When a program uses templates, ClearCase often associates incorrect configuration information with the program file and the files in the repository.

Models for Working with IBM XLC

To help **clearmake** and the XLC C++ compiler work together effectively, we recommend that you set up your makefile and program sources according to one of the models described in this section. The model you choose depends on your needs:

- The Simple model is easy to use and solves some of the problems mentioned earlier. It is the only model that makes use of automatic template instantiation to build template code efficiently. It works best for developers who work independently of others. It is not as effective for team projects in which sharing builds is important.
- The Compile-Time Demand Instantiation model requires work to set up, but it is easy to use and avoids all the **clearmake** problems mentioned earlier. It also simplifies sharing builds in a team project. The penalty of using this model is that compile time can increase.
- The Explicit Instantiation model is difficult to use, because it requires you to track template references. This model avoids all the **clearmake** problems mentioned earlier and simplifies sharing builds in a group project. It is also the best model to use for building archives and shared libraries that contain template code.

The remainder of this section describes each model in more detail.

The Simple Model

The Simple model relies on the automatic template instantiation of the XLC compiler. This model requires that you insert one line into your makefile.

The Simple model does not solve all the problems that result from conflicts between XLC automatic template instantiation and **clearmake**, but it allows you to make minimal changes to the project and to take advantage of the efficiency of automatic template instantiation. Configuration problems can result from version changes in the template sources. The Simple model is not appropriate for building libraries.

Modifying the Source Files

The XLC automatic template instantiation scheme requires that for every .h template header file, there is a corresponding .c source file with the same name that contains the implementation of the template. The .c extension is unusual in C++ source file names; you may want to set up your project source files so that each .c file refers to a corresponding source file with a more standard extension, such as .cxx or .C. You can set up a reference .c file in two ways:

- Create a new element with the **.c** extension that includes the corresponding file of the standard extension.

- Create a link with the **.c** extension that points to the corresponding file of the standard extension. For instructions about how to create a link in an MVFS directory, see the **cleartool ln** reference page.

Note: Do not set up template header files to include the corresponding source file. Doing so disables automatic template instantiation and defeats the Simple model.

Designing Your Makefile

To design your makefile:

1. Insert the following line into your makefile, at any point in the file that does not break a rule or definition that spans multiple lines.

   ```
   .DEPENDENCY_IGNORED_FOR_REUSE: tempinc/%.C
   ```

   This directive instructs **clearmake** not to rebuild program components that depend on a compiler-generated source file in the repository when that file changes or is deleted. Such rebuilds are unnecessary. This directive works if the leaf name of the repository directory is tempinc.

2. If you use the **–qtempinc=tempinc** compiler option to rename a repository directory, add to your makefile another directive of the form

   ```
   .DEPENDENCY_IGNORED_FOR_REUSE: my_repository/%.C
   ```

   where *my_repository* is the leaf name of the repository.

Limitations of the Simple Model

Some problems remain when you use the Simple model. The following list describes the problems and how to work around them.

- After you modify a template source file, **clearmake** sometimes fails to relink the executable.

  To work around this problem, delete the executable before you build with a new version of a template source file in your program.

- When you revert to a version of a template source or header file that is older than the version the compiler used in a previous build, the compiler may reuse object modules in the repository that it should have recompiled. This can happen after you change a template, a checked-out source, or a header file from reserved to unreserved or after you modify your configuration specification. **clearmake** detects the change and recompiles the file. However, because the compiler relies on a file time-stamp comparison of cached template code with its source, it incorrectly reuses the code already in the repository.

To work around the problem, remove all .o files from the repository when you revert to an older version of a template source or header file.

- **clearmake** rebuilds program components rather than winking in available builds from another view. This problem makes the Simple model more difficult to use for team projects in which sharing builds is important.

  To manually wink in files produced by another build, use the **cleartool winkin** command. Use the –**recurse** option of the **winkin** command to wink in a hierarchy of files that produced a build. For more information, see the **winkin** reference page.

- The configuration record for the final program contains confusing information about the dependency structure of the program. This problem becomes evident when you execute **cleartool** commands such as **catcr** and **diffcr**, which display configuration records.

The Compile-Time Demand Instantiation Model

The Compile-Time Demand Instantiation model does not use the XLC compiler's automatic template instantiation. Instead, it forces the compiler to instantiate all the template code used by a given module. At link time, there may be multiple definitions of template symbols, but the linker drops the duplicate symbols.

To use this model, you may have to edit some source files. The advantage of using this model is that avoiding automatic template instantiation solves the conflicts between the compiler and **clearmake**. The disadvantage is that compiling duplicate template code takes extra time and disk space.

Modifying the Source Files

The Compile-Time Demand Instantiation model requires that every template header file include its corresponding source file. If you use this model, you must edit template header files that do not include their corresponding source file.

We recommend that you use conditional compilation directives as shown in the following pattern.

```
// Header file my_templ.h
//
#ifndef _my_templ_h__ // header file exclusion -- use any
                      // unique name
#define _my_templ_h__

... Place template declarations here ...
```

```
#ifdef TEMPLATE_CODE_IN_HEADERS
#include "my_templ.cxx" // source file name suffix may vary
#endif
#endif // _my_templ_h
```

Designing Your Makefile

If you set up your header files as recommended in the previous section, you must also define the macro that enables source file inclusion on the compiler command line. For the sample above, enable source file inclusion by specifying the compiler option **–DTEMPLATE_CODE_IN_HEADERS**. The conditional directive allows you to build under a different model or on other platforms without changing the source code.

To disable automatic template instantiation, specify the **–qnotempinc** compiler option.

Duplicate Symbol Warnings from the Linker

When you set up template header files to include their source files, the implementation of each template is available to the compiler at compile time. The compiler instantiates the template immediately and puts the compiled template code into the object module. As a result, every object module that refers to a certain template function or data contains the code for that template function or data, which leads to multiple definitions of template symbols at link time. The AIX linker drops the duplicate symbols without generating an error. As the linker drops each duplicate symbol, it issues a `Duplicate symbol` warning. You can ignore these warnings. To suppress them, specify the compiler option **–LOOK_IT_UP** in the final link step of the program.

The Explicit Instantiation Model

The Explicit Instantiation model is useful whenever it is necessary to assume full control over the placement of instantiated template code. For example, if you are building an archive or creating a shared library that contains template code, you probably want to use this model. To use the Explicit Instantiation model, you must disable automatic template instantiation and on-demand compile-time instantiation, and explicitly identify the templates that your program needs to instantiate.

The Explicit Instantiation model requires that you keep track of the set of templates the program requires and maintain source directives to instantiate the templates. This maintenance burden is worthwhile only when you are required to specify the module in which instantiated template code appears, for example, when building code for an archive or shared library. Otherwise, the Compile-Time Demand Instantiation model is more suitable.

Modifying the Source Files

For each template class or function that your program requires, add exactly one **#pragma define** directive to the source code. For example, if your program requires the class

```
Array<String>
```

then add this **#pragma define** directive:

```
#pragma define(Array<String>)
```

In each source file that contains a **#pragma define** directive, include the source files that contain the implementations of the templates used. For example, if the source for the Array template is contained in the file array.cxx, add the following **#include** directive to the file that contains the **#pragma** directive:

```
#include "array.cxx"
```

Do not set up header files to include corresponding source files. If you do, duplicate template instantiation may result (see *The Compile-Time Demand Instantiation Model* on page 125).

Remove function definitions from source files that contain **#pragma define** directives.

Designing Your Makefile

In your makefile, disable automatic template instantiation by specifying the **–qnotempinc** compiler option.

Working with the HP aC++ Compiler

This section describes how to use **clearmake** to build C++ programs using the HP aC++ compiler (**aCC**).

If you are using the earlier HP Cfront-based compiler (**CC**), read *Working with Cfront-Based C++ Compilers* on page 91 instead of this section.

The HP aC++ compiler offers several methods of building template code; the method you choose depends on your needs. The following sections describe these methods.

Automatic Instantiation

By default, the HP aC++ compiler instantiates templates automatically to build template code. During a prelink step, the compiler determines what template code the program requires and compiles the necessary code into some of the object files that make up the program. The compiler tracks how the program uses template code. There

is no repository; instead, the compiler records this information in .o and .l files in the build directory.

If you use the Automatic Instantiation method, **clearmake** sometimes rebuilds program components unnecessarily. When the prelinker compiles template code into an existing object file, the dependency information that **clearmake** previously recorded for that object file is no longer up to date. The next time **clearmake** is invoked, it rebuilds the object file; the dependency information for the object is again correct. At this point, **clearmake** no longer rebuilds that object file unnecessarily.

The Automatic Instantiation method is the easiest method to use because it requires no programmer intervention. It is suitable for most applications. However, it does cause occasional unnecessary rebuilds.

Command-Line Option Instantiation

The HP aC++ compiler provides various command line options for specifying what templates to instantiate for a given translation unit. For example, the **+inst_all** option requests instantiation of all templates and the **+inst_used** option requests instantiation of templates that are used. With these options, the compiler compiles all template code into the object module. If multiple source modules refer to the same template class or function, copies of the compiled template code appear in multiple object modules.

The command-line instantiation methods are easy to use; they require only that you specify extra compiler options. These methods are suitable for most applications and do not conflict with **clearmake** build avoidance. The disadvantage of using these methods indiscriminately is that the compiler requires extra time and disk space to perform redundant template instantiation. Indiscriminate use may also result in duplicate symbols.

Note that a special command-line option, **+inst_close**, is provided for building archives and shared libraries. With this option, the compiler uses the automatic instantiation method.

Explicit Instantiation

The HP aC++ compiler supports the ANSI C++ explicit instantiation syntax as defined in the ISO *Standard for the C++ Programming Language*. This explicit instantiation method requires you to request instantiation explicitly in the source code. For more information about using explicit instantiation, see the HP aC++ compiler documentation.

The Explicit Instantiation method requires more effort to use; however, it allows you to control the placement of instantiated template code into object modules. In addition,

as part of the ANSI C++ standard, it may be the most portable solution. Using explicit instantiation does not conflict with **clearmake** build avoidance.

Using ClearCase Build Tools with Java

8

The build behavior of Java tools causes various problems for **clearmake**. This chapter presents these problems and some possible solutions.

The standard Java toolkit is the Java Development Kit from Sun Microsystems. It includes the **javac** compiler and is available for many platforms.

The remainder of this chapter uses **this compiler** as the example.

Using make Tools with javac

Although **javac** handles dependency analysis well, using **javac** by itself misses rebuilds that a **make** tool does not. Specifically, **make** detects modifications of indirect dependencies that **javac** does not. If A.java depends on B.java and B.java depends on C.java, when you change C.java, the command **javac A.java** does not rebuild C.class. Therefore, if you are using **javac** directly, you must recompile each file as you change it.

Many Java applications have some components that are compiled natively or are written in another language. For at least those parts of their applications, developers need makefile-based building.

Using javac with clearmake

There are additional benefits of using **clearmake** instead of **make**, especially given the building behavior of **javac**. **clearmake** is better at determining when a rebuild is required than are the Java tools, with or without **make**.

When using dynamic views, **clearmake** detects the following rebuild cases that **javac** does not:

- Selection of an older version of a .java file. Because the rebuild decision is based on an older/newer comparison, **javac** does not detect that a rebuild is necessary.

  Note: Clock skew between hosts can cause similar time stamp problems outside ClearCase.

- Change of the **javac** command line. If the command-line options used to build a .class file have changed since the last build, **clearmake** rebuilds the .class file. For example, if you add the **–g** option to direct the compiler to rebuild with debugging information, you must invoke the compiler on all your .java files to ensure that they are rebuilt to contain the debugging information.
- Manual winkin of a .class file that is out of sync with, but newer than, the corresponding .java source selected by the view. Because the rebuild decision is based on an older/newer comparison, **javac** does not detect that a rebuild is necessary.

However, some problems may arise when using ClearCase build tools to perform Java builds.

ClearCase Build Problems with javac

ClearCase problems with Java builds relate to conflicts between the dependency analyses of **clearmake** and **javac**. **clearmake** is designed to control the build order and the dependency checking between parts of a build system. Java compilers are also designed to provide some of this functionality. When **clearmake** audits build tools that perform their own dependency checking, the result can be unnecessary rebuilds, builds that are never considered up to date, or derived objects that are not winked in. Consequently, the process of managing Java builds with **clearmake** can sometimes produce unexpected results.

For example, if makefiles declare all .class file dependencies accurately, **clearmake** can run the compiler in a sequence such that the compiler never builds more than is necessary. But if .class file dependencies are not declared, the compiler may build files that **clearmake** did not ask it to build. These inaccurate declarations may cause **clearmake** to treat .class targets as out of date, even if the compiler treats them as current. In some cases, **clearmake** may treat a build system as if it is never current. Moreover, the process of maintaining Java makefiles that accurately declare all .class dependencies is both difficult and tedious.

To solve ClearCase build problems with **javac**, you can use a special target called **.JAVAC**.

Using the clearmake makefile Special Target

The **.JAVAC** target enables **clearmake** to use heuristics on audits of Java builds to accurately infer .class dependencies. These dependencies are then stored in .class.dep files, enabling future **clearmake** runs to build .class targets such that each compiler invocation builds only one .class file.

Using .JAVAC causes **clearmake** to take the following actions:

- Examines the audit of any target with the file name extension .class
- Infers the class dependencies from the relative order of .java and .class files that the compiler accesses
- Records these dependencies in .class.dep files
- In subsequent builds, reads .class.dep files to augment the makefiles and build the .class targets in an order that agrees with the compiler's dependency checking and precludes it from building anything not specified on the command line.

Unsupported Builds

clearmake does not support the use of the **.JAVAC** target with nonaudited or parallel builds. The **.JAVAC** special target requires both a dynamic view context and audited builds.

When **clearmake** begins to build two targets in parallel, it does not account for the possibility that two targets may write common files. If they do, the builds may collide when writing those files.

If **.JAVAC** is used with parallel builds, **clearmake** prints the following message:

```
clearmake: Warning: Use of .JAVAC with parallel builds implies
.NOTPARALLEL; ignoring -J
```

Makefile Requirements for .JAVAC

The **.JAVAC** special target must be used with no dependencies and no build script:

```
.JAVAC:
```

clearmake issues a warning and ignores any dependencies or build script.

Other than that, makefiles must use implicit (suffix or pattern) rules. For example:

```
.SUFFIXES: .java .class
.java.class:
        rm -f $@
        $(JAVAC) $(JFLAGS) $<
```

Makefiles may also use pattern rules for compatibility modes that support them. For example:

```
%.class: %.java
        rm -f $@
        $(JAVAC) $(JFLAGS) $?
```

The makefiles must also use absolute paths for .class targets. For example:

```
all_classes: /vobs/proj/src/pkg1/foo.class
```

clearmake contains a built-in macro function you can use to specify absolute paths more easily:

```
$(javaclasses)
```

For more information about the javaclasses macro, see *Using the javaclasses Built-in Macro*.

The first time **clearmake** builds with **.JAVAC** enabled, **clearmake** is not making use of any inferred knowledge before it builds the first .class target.

After the first build session completes, the .class dependencies are known. As a result, some targets may be rebuilt in the second build session in that view, because **clearmake** is now using the .class.dep files, which may declare .class dependencies that require a different build order. After the second build session completes, the .class DOs are reusable in that view.

Using the javaclasses Built-in Macro

The $(**javaclasses**) function can be used to generate a list of .class targets, from existing .java files. The function takes one or two path arguments.

If your builds write .class files to the same directory as the .java files, use the single-argument form. For example:

```
ALL_CLASSES=$(javaclasses .)
```

If this makefile resides in /vobs/myvob/src, along with A.java and B.java, then the previous command sets ALL_CLASSES to

```
/vobs/myvob/src/A.class /vobs/myvob/src/B.class
```

To provide a way for **clearmake** to evaluate all the .class targets, use this macro in the dependency list of some other target. For example:

```
ALL_CLASSES=$(javaclasses .)
  all : $(ALL_CLASSES)
```

If your builds writes .class files to a different directory from the .java files, use the two-argument form. The first argument is the destination root, an absolute path beneath which your package .class files are written. The second argument is the source root, an absolute path beneath which your package sources reside.

If this makefile resides in /vobs/myvob/src/pkg1, along with files A.java and B.java, and you want the compiler to write the .class files into /vobs/deployvob/pkg1, use a function like this one:

```
ALL_CLASSES=$(javaclasses /vobs/deployvob, /vobs/myvob/src)
```

This sets ALL_CLASSES to

```
/vobs/deployvob/pkg1/A.class /vobs/deployvob/pkg1/B.class
```

A Java compiler that is run from the source directory for Package1 can compile .java files stored in the source directory for Package2. **clearmake** can manage this process if the makefile can always be used to find a rule to build a .class target, regardless of where the target's .java file resides.

To this end, we recommend that you use suffix rules in your makefiles. For example:

```
.JAVAC:
.java.class:
        rm -f $@
        javac $<
```

When using .JAVAC, **clearmake** can use this type of rule to build any .class file and correctly set the value of built-in macros like $@ and $<.

For example, if this makefile and the A.java file reside in /vobs/myvob/src/pkg1, you can instruct the compiler to write the .class files to /vobs/deployvob by using a makefile fragment like this one:

```
.JAVAC:
DEPLOY_ROOT=/vobs/deployvob
  ALL_CLASSES=$(javaclasses /vobs/myvob/src, $(DEPLOY_ROOT))
  all: $(ALL_CLASSES)
.java.class:
        rm -f $@
        javac -d $(DEPLOY_ROOT) $<
```

Deriving Class Dependencies

clearmake stores dependencies it derives in .class.dep files. Each .class target which produces .class DOs has a .class.dep file. For example, for a target named A.class, **clearmake** creates an A.class.dep file. **clearmake** reads .class.dep files when evaluating a target to augment its makefile-declared dependencies.

clearmake derives class dependencies as follows:

- **clearmake** records only direct dependencies in .class.dep files.

  If class A uses B uses C, then **clearmake** records in A.class.dep that A.class depends on B.class, and in B.class.dep that B.class depends on C.class. The transitive dependency of A.class on C.class is implicit in that pair of dependencies and is not explicitly stated in any .class.dep file.

Storing Class Dependencies

clearmake retains derived class dependencies by writing them to .class.dep files, one per .class target, to be used by future **clearmake** build sessions as explicit dependencies because they determine the order of target evaluations.

The .class.dep files are in XML format.

Java Cyclic Class Dependencies

When **clearmake** detects a new .class dependency cycle, it marks the targets as being part of a target group. After **clearmake** forms a target group for a cyclic class dependency, the existing .class DOs from the new target group cannot be used in the view because the build script for those DOs has changed. The build script difference causes the next **clearmake** invocation to rebuild all three DOs.

Using .class.dep Files

clearmake reads the contents of a .class.dep file when it evaluates the dependency list of the corresponding .class target. For example:

```
A.class: A.java
        rm -f A.class
        javac A.java
```

For this makefile, **clearmake** tries to find, open, and parse A.class.dep before it evaluates A.java. The stored dependencies, if any, are evaluated before A.java.

Because .class.dep files are DOs, **clearmake** can wink in these files from other views. The .class.dep files have their own config record, which has a single dependency on the corresponding .java version, and no build script. If two views can compile the same .java version with different compiler options, and the same class dependencies are derived, a .class.dep file from one view could be winked in to the other.

Note that .class.dep and .class files have no config record relationship.

Dependencies and DO Reuse/Winkin

We recommend that you always delete .class.dep files and .class files together. Avoid deleting only one of these types of files while leaving the other type.

If A.java has been changed, **clearmake** audits a rebuild of A.class to determine whether the A.class.dep file's dependencies have changed. If they have not, A.class.dep can be reused. **clearmake** does not attempt to reuse .class.dep files as DOs.

Note that whenever a .class.dep file can be found in the view, **clearmake** does not shop for it in another view.

clearmake cannot wink in a .class DO if it does not know all of the class dependencies with which the DO was built.

Effects of Nested Classes

A Java class can be declared inside another class. For example:

```
class A {
    class inner {
        [...class definitions...]
    }
    [...class definitions...]
}
```

The compiler writes the code for nested classes into a separate .class file, whose file name is <*parent_class_name*> $ <*nested_class_name*>. In the previous example, the compiler writes A.class and A$inner.class.

clearmake does not create .class.dep files for inner classes. The .class.dep files are intended to declare only the dependencies of one .java file's classes on another .java file's classes. Inner .class files are always siblings. The use of anonymous inner classes, which have a compiler-generated name, is handled in the same way.

.JAVAC in BOS Files

Users can put the .JAVAC special target in build options specification (or BOS) files such as in Makefile.options, rather than in Makefile.

.SIBLINGS_AFFECT_REUSE

By default, **clearmake** does not check sibling DOs when determining whether a target DO in the view can be reused. Using .JAVAC causes **clearmake** to behave as though .SIBLINGS_AFFECT_REUSE was specified in the makefile.

Building Java Applications Successfully without .JAVAC

The following alternatives allow you to successfully build Java applications with **clearmake**:

- Write the makefile correctly
- Allow **clearmake** to rebuild

- Configure **clearmake** makefiles to behave like **make**

The following sections describe each option in detail.

Writing Correct Makefiles

A correctly written makefile results in a correct set of configuration records, which gives you the full power of ClearCase configuration records and winkin without unnecessary rebuilding and without missing rebuilds. You can restructure a makefile to avoid **javac**'s automatic building behavior by enforcing that files on which other files depend are built before their dependents.

Note: clearmake detects implicit dependencies but cannot determine build order dependencies. If you want files to build in a certain order, you must declare that order in the makefile by adding additional dependencies.

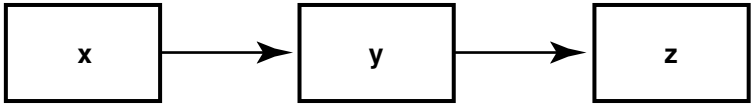
You must take extra care when handling mutually dependent files, because there is not necessarily a correct order for building them. One possibility is to always generate all mutually dependent files as one unit, that is, in one configuration record. You can write the build script for a set of mutually dependent files to delete all class files that correspond to those files before building any of them. This practice ensures that they are not overwritten individually and makes them available as a unit for winkin.

The advantage of writing your makefile correctly is that you avoid extra compilations or rebuilds. No special makefile directives are required, the configuration records have no unusual properties, and winkins work fully. The disadvantage is that the makefile must always be synchronized with the structure and dependencies of the application.

The following sections are makefile examples for applications with particular dependency characteristics.

No Mutually Dependent Files

In this application, classes x, y, and z have a hierarchical dependency graph:



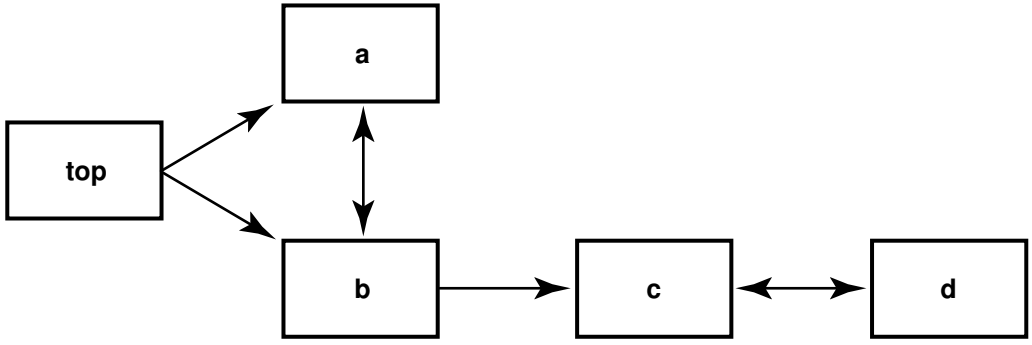
The makefile for such a dependency structure is very simple:

```
.SUFFIXES: .java .class

.java.class:
       javac $<
x.class: y.class
y.class: z.class
```

Mutually Dependent Files

This application consists of classes **top**, **a**, **b**, **c**, and **d**, which have a more complex dependency structure:



The makefile for this dependency structure is somewhat longer, but correct:

```
top.class: a.class b.class
       javac top.java

a.class: b.class

b.class:
     rm -f a.class b.class
     javac a.java b.java

b.class: c.class

c.class: d.class

d.class:
     rm -f c.class d.class
     javac c.java d.java
```

Allowing Rebuilds

If you continue to invoke **clearmake** until it determines that all files are up to date, other ClearCase features work correctly. The configuration records record all .class files as implicit dependencies rather than as siblings, which allows winkin to work.

However, the number of rebuilds can become very large if the makefile is written incorrectly. It is possible to map out a correct set of dependencies, as described in *Writing Correct Makefiles* on page 138; it is also possible to request (however inadvertently) that **clearmake** build the files in exactly the reverse, and most inefficient, order.

In addition, **clearmake**'s default behavior is to ignore modifications to siblings for the purposes of rebuilding. For winkin to work correctly, you must reenable that behavior by using a command-line option or special makefile directive.

Another drawback to this method is that the builds of mutually dependent source files do not fit well, because the files are never up to date. The makefile for these must be written carefully, as described in *Writing Correct Makefiles* on page 138.

Configuring Makefiles to Behave Like make

By using special targets, you can configure your **clearmake** makefile so that **clearmake** behaves as **make** does with regard to Java builds. The following targets eliminate the extra rebuilding described in *Allowing Rebuilds* on page 139:

```
.NOCMP_SCRIPT: %.class              (omake only)

.NO_CMP_SCRIPT: %.class             (clearmake only)

.DEPENDENCY_IGNORED_FOR_REUSE:
%.class
```

.NO_CMP_SCRIPT disables build script checking. However, relevant build-script changes are ignored. In addition, **.NO_CMP_SCRIPT** has no effect during winkin, so even when it is in use, winkins are prevented because of build script differences. Therefore, you must use manual winkins (see the **winkin** reference page) or forego them entirely.

.DEPENDENCY_IGNORED_FOR_REUSE disables the version checking of implicit dependencies when **clearmake** is looking for DOs to reuse. This can cause desired rebuilds to be missed, however. One benefit of using **clearmake** is automatic dependency detection (for example, of .h files in a C build), so it is not desirable to give this up.

To improve the missed implicit dependency checking caused by **.DEPENDENCY_IGNORED_FOR_REUSE**, you can add the missing dependencies as explicit dependencies in the makefile. However, this is a manual process, and you still lose build script checking and winkin. The remaining benefit of using **clearmake** is configuration records (though the **catcr** output for them may be confusing).

Setting Up a Parallel Build

9

This chapter describes the process of setting up and running builds that use several hosts in the local area network. It describes the control mechanisms for the client and server.

Overview of Parallel Building

Rational ClearCase can perform builds in which multiple processes execute in parallel the build scripts associated with makefile targets. The processes executing the build scripts can run on a single host or across a collection of machines around the local area network (parallel distributed build). By using more concurrent processes, parallel builds can reduce the overall build time significantly. Instead of one process running one build script at a time, you can have multiple processors working in parallel. For large software systems, this performance improvement can make a critical difference. For example, you can use parallel builds to enable a build of your entire software system to run overnight and finish before developers and testers arrive for work in the morning.

You start a parallel build the same way as a single-host build: by entering a **clearmake** command. A command-line option or environment variable setting causes the build to run in parallel mode.

A parallel build is controlled by specifications on all the hosts involved. The host on which you enter the **clearmake** command is the *build client* or *build controller*. On this host, you specify a limit to the number of build scripts to be executed concurrently. You can also specify a build hosts file, which lists the server hosts to be used for building.

Each build server host used in a parallel build can have an access-control file, named bldserver.control. To use the host as a build server, a build client must meet the access-control requirements.

When building in parallel, **clearmake** starts one or more audited build executor (**abe**) processes. An **abe** is a server process invoked by **clearmake** to control and audit execution of a build script during a parallel build. The first time it dispatches a build script to a host, **clearmake** starts an **abe** process there. Subsequent build scripts

dispatched to the same host may be executed by the same **abe** process or by a different one.

An **abe** process starts by setting the same view as the calling **clearmake**. It executes a build script dispatched to it in much the same way as **clearmake**—each command in a separate shell process.

All **make** macros are expanded by the build script calling **clearmake**, but environment variables are expanded by the shell process in which a build command runs. This environment combines the **abe** startup environment and the entire environment of the calling **clearmake**. Where there are conflicts (for example, SHELL and PATH), the **abe** setting prevails. To this environment other macros are added:

- Special make macros, such as **MAKEFLAGS**, **MAKEARGS**, and (in the compatibility modes **sgismake**, **sun**, and **gnu**) **MFLAGS**. These are needed in case the build script invokes **clearmake** recursively.
- Macros assigned in a BOS file or on the **clearmake** command line. These settings are always placed in the build script's environment; they override, if necessary, settings in the environment of the calling **clearmake** or settings in the **abe** startup environment.

The **stdout** and **stderr** output that produce build scripts is sent back to **clearmake**, which stores it in a temporary file. When the build script terminates, **clearmake** prints its accumulated terminal output.

abe returns the exit status of the build script to the calling **clearmake**, which indicates whether the build succeeded or failed. If the build succeeded, **abe** creates derived objects and configuration records.

Note: The **abe** program is started by **clearmake** when needed. Never run it manually.

Each **abe** process sets to the same view and working directory as the **clearmake** process; each process then executes build scripts dispatched to it from the controlling **clearmake** process. When the **abe** process is being started on a remote machine, **clearmake** uses the standard UNIX remote shell facility, referenced through the *ccase-home-dir***/etc/rsh** symbolic link. A build script runs under **abe** control as if it were executed by **clearmake**, except that **abe** collects terminal output produced by the build script and sends it back to the build controller, where it appears in your **clearmake** window. The **abe** process terminates after waiting three minutes for an initial connection or after waiting three hours for a subsequent response.

Parallel Build Scheduler

clearmake schedules and manages target rebuilds as follows:

- It executes the build script for an out-of-date target as soon after detection as system build resources allow.
- It does not assume that executing a build script for a specific target implies that the target was updated.

clearmake evaluates the dependency graph, beginning with the targets specified on the command line. Before evaluating a specific target, **clearmake** ensures that all dependents of that target have been evaluated and brought up to date. As soon as a target is deemed to be out of date, it is made available for rebuilding. A rebuild is initiated as soon as system resources allow. Depending on the availability of build hosts and load-balancing settings, this may happen immediately or be delayed.

When DO shopping/winkin occurs, **clearmake** postpones DO lookup for any target that has scheduled dependents until the target is encountered in the rebuild logic. When such a target is detected, **clearmake** then attempts the DO shopping/winkin only when the target's dependencies have completed. This delay eliminates unnecessary rebuilds in serial mode and allows a parallel **clearmake** to initiate rebuilds sooner.

Failure Modes

Certain conditions can interfere with an **abe** process, causing a target rebuild to fail:

- Remote login is disabled on a particular host, preventing an **abe** process from being started.
- **clearmake**'s view could not be accessed on the remote host.

Setting Up the Client Host

There are three issues to consider when you set up client-side processing for parallel builds:

- The number of parallel build processes to request. **clearmake** limits the number of concurrent target rebuilds to the value of the CCASE_CONC environment variable or the number that you specify with the **–J** command-line option. **clearmake** does not start more **abe** processes than specified.
- The hosts to specify for builds. **clearmake** starts all **abe** processes on the local host unless you provide a build hosts file. You can specify a build hosts file with the **–B** command-line option or with the CCASE_HOST_TYPE environment variable.
- The limitations and requirements for system loading on the build hosts. ClearCase tries to keep the build machines from becoming overloaded by dispatching a build

script to a host only if the host is sufficiently idle. The idleness threshold is specified in the build hosts file. If you do not specify an idleness threshold or do not use a build hosts file, **clearmake** submits as many build scripts as allowed by the **–J** or CCASE_CONC setting.

Creating Build Hosts Files

The build hosts file is the client-side control file for parallel builds.

Build hosts files that you specify by using the CCASE_HOST_TYPE environment variable must be located in your home directory, and each file must have a name that begins with .bldhost. Choose a file name extension for each build hosts file that describes its intended use. For example:

.bldhost.sun5	List of hosts used to build SunOS 5 binaries
.bldhost.day	List of hosts used to perform parallel builds during the workday
.bldhost.night	List of hosts used to perform overnight parallel builds

Build hosts files that you specify with the **–B** option can be located anywhere and do not have to have special names.

Your build environment determines whether you need multiple build hosts files. In a heterogeneous network, for example, architecture-specific builds may or may not need to be performed on hosts of that architecture. (You may have cross-compilers, which eliminates this restriction.)

When you start a parallel build, you can specify a certain build hosts file or have **clearmake** select one by using the **–B** option. If you do not specify **–B** when running a parallel build, **clearmake** does the following:

1. Determines the host type.
2. Looks in the password database to determine your home directory.
3. Uses the file .bldhost.$CCASE_HOST_TYPE in your home directory.

For example, you can set up two build hosts files, for daytime and nighttime use, as follows:

1. Create a build hosts file for daytime use. For daytime builds, you can use the list of hosts that your system administrator has provided in /usr/local/lib, along with your own host. To minimize the disruption to other work, you can specify that each host is to be used only if it is not heavily loaded, that is, if it is at least 75% idle.

```
% cat > $HOME/.bldhost.day
-idle 75
neptune
#include /usr/local/lib/day_builds
<CTRL+D>
```

2 Create a build hosts file for overnight use. For overnight builds, you can use another list of hosts provided by the system administrator.

```
% cat > $HOME/.bldhost.night
#include /usr/local/lib/night_builds
<CTRL+D>
```

Because this file does not include an **–idle** specification, **clearmake** uses a host only if it is at least 50% idle.

If CCASE_HOST_TYPE is set, but **clearmake** cannot find or read the build hosts file, it does not perform the build. (**clearmake** assumes that if CCASE_HOST_TYPE is set, you want to perform a parallel build. Because the parallel build may use a host with a different architecture than the local host, performing the build on the local host may yield incorrect results. Therefore, by default, **clearmake** does not build on the local host if CCASE_HOST_TYPE is set.)

clearmake does not use a host during a parallel build if your current view cannot be used on that host. (For example, the host may not be able to access the view's storage directory.)

We recommend that you set the CCASE_HOST_TYPE variable conditionally in your makefile, using target-dependent variable bindings. If you set the variable on the **clearmake** command line, in your process environment, or unconditionally in your makefile, it applies to all targets.

Note: clearmake supports target-dependent variable bindings in standard mode and in Sun compatibility mode. You can also use target-dependent variable bindings in your BOS file for any compatibility mode.

For example, to ensure that the target **x** is built on host **neon** or **saturn**:

```
x := CCASE_BLD_HOSTS = neon saturn
```

You can also use patterns in target names. For example, to build all .o files on host **pluto**:

```
%.o := CCASE_BLD_HOSTS = pluto
```

clearmake applies CCASE_BLD_HOSTS bindings to dependencies of the specified targets. To apply CCASE_BLD_HOSTS to the specified targets but not their dependencies, add the line shown below to the built-ins file for your compatibility mode:

Mode	Location of builtins file	Line to add
standard	*ccase-home-dir*/**etc/builtin.mk**	`% := CCASE_BLD_HOSTS =`
Sun	*ccase-home-dir*/**etc/sunbuiltin.mk**	`% := CCASE_BLD_HOSTS =`

To determine the name of any .bldhost file **clearmake** reads during the build process, use the **clearmake -v** command.

Load Balancing

The ClearCase load-balancing algorithm controls the way in which build scripts are dispatched to hosts. During a parallel build, your **clearmake** process creates and updates a list of qualified hosts, a subset of the hosts listed in the build hosts file. A host is qualified if all these criteria are met:

- The host is at least 50% idle.
- Your **clearmake** process meets the host's requirements, as specified in its bldserver.control file.
- An **abe** process can be started on the host.

Whenever it needs to dispatch a build script, **clearmake** updates its qualified hosts list and selects one of these hosts. If it cannot find any qualified host, it pauses and updates the list again. (On any pass, if all hosts are eliminated because of errors, **clearmake** exits. If the hosts do not meet the first two requirements, **clearmake** waits and tries again.) **clearmake** keeps trying in this manner until it finds at least one qualified host with which to build.

The selected host is not necessarily the best one, for example, the one that is most idle at that particular moment.

Randomizing Host Selection

The default load-balancing algorithm tends to select hosts near the top of the list more often than those near the bottom, subject to availability. For more even-handed selection when the list of hosts exceeds 20 or so, include this line:

```
-random
```

Note that this also changes the effective location of any **#include** directives.

A **–random** line can appear anywhere within a build hosts file. It applies to all host names in the build hosts file.

Idleness Threshold

By default, your **clearmake** process does not dispatch a build script to a host unless it is at least 50% idle. You can adjust this idleness threshold with a line in the build hosts file:

–idle *percentage* [%]

percentage can be any integer from 0 to 100. Idleness is negatively correlated with the host's load factor, as shown by **uptime(1)**; the approximate correspondence is this:

Load	Idle Percentage
0.0	100
0.5	68
1.0	47
2.0	22
4.0	almost 0

–idle directives can appear anywhere within a build hosts file. Until a **–idle** directive appears, the default value of 50% is applied.

–idle can appear multiple times within a build hosts file. Each **–idle** directive applies to the host names that follow, until another **–idle** directive appears or the end of the build hosts file is reached.

Because a host can appear multiple times in a build hosts file, it can be associated with different idleness thresholds. Each inclusion of the host is recorded with the associated idleness threshold. (In practice, if a host appears twice, once with a low idleness threshold and once with a high threshold, **clearmake** may select it once but reject it another time.)

Note: The idleness threshold can be specified with **–idle** directives on both the client and server. If there is a conflict, the overall principle is that the build server host controls its fate. For example:

- A **clearmake** process is searching for hosts that are at least 50% idle (the default). A build server that appears to qualify because it is 70% idle is not used if its bldserver.control file includes the line **–idle 75**.
- A bldserver.control file on a build server host permits access, because it contains the line **–idle 60** and the host is currently 75% idle. However, **clearmake** does not dispatch a build script to this host, because the build hosts file specifies a higher threshold: **–idle 80**.

clearmake uses the idle specification in your host's bldserver.control file to determine whether it can perform the build on your host. If your host does not have a bldserver.control file, **clearmake** assumes an idle threshold of zero and performs the build regardless of the load on your host. If a specified host appears in your build hosts file, **clearmake** ignores any **–idle** specifications for the host in the build hosts file and uses **–idle 0**.

Include File Facility

A build hosts file can include the contents of one or more other build hosts files:

```
#include pname
```

If the included file has **–random** or **–idle** directives, they apply to that file's entries. A **–idle** directive in a file is passed down to included files, until and unless it is overridden by another **–idle** directive. (A **–idle** directive in an included file does not affect the including file.)

Any line in the include file that begins with a number sign (**#**) (except an **#include** line) is treated as a comment.

Note: ClearCase evaluates environment variables in *pname* during builds.

Including Comments in a File

You can include a comment on a line by itself or at the end of a hostname line. Comment lines must begin with **#** and end with a <NEWLINE>. For example:

```
# Solaris build hosts
#
neon    # cpu Sparc, 150 MHz, Avail Mem 64MB
silicon # cpu Sparc, 400 MHz, Avail Mem 256MB
```

With the exception of **#includ**e, a number sign (**#**) always indicates the start of a comment and **clearmake** ignores the rest of the line.

Note: You cannot put comments on **–idle**, **–random**, or **#include** lines.

Examples

- Build hosts file that uses a listed host only if it is at least 75% idle:

  ```
  -idle 75
  mercury
  earth
  mars
  pluto
  ```

- Nesting of build hosts files:

```
-idle 30
einstein
bohr
fermi
#include /usr/local/lib/planet.hosts
```

- Use an environment variable to specify where other build hosts files are located.

  ```
  #include ${BLDHOSTPATH}/build_hosts
  ```

- Use multiple **–idle** directives to control build access.

  ```
  # my machines
  -idle 10
  neon
  saturn
  # project build hosts
  #include ${BLDHOSTPATH)/dev_build_hosts
  # other random hosts
  -idle 50
  sunfast
  bigzilla
  ```

  The project build hosts file looks like this:

  ```
  # The development project owns these machines
  -random
  chirp
  mew
  -idle 10
  growl
  roar
  ```

Setting Up Trust Relationships

For parallel building to work correctly, the client host must be trusted by the build hosts (that is, remote login from the client host to the build hosts must work without a password being needed). You can set up this trust with .rhosts files on the hosts, or by having your system administrator set up general trust (for example, with /etc/hosts.equiv). To test remote login, execute the following command (substituting your ClearCase installation directory for *ccase-home-dir*):

*ccase-home-dir***/etc/rsh** *remote-hostname* **echo Good-to-use**

If you are prompted for a password, the client host is not trusted.

For example:

/usr/Rational/etc/rsh puffin echo Good-to-use

```
Good-to-use                          (current host is trusted by host puffin)
```

/usr/Rational/etc/rsh auk echo Good-to-use

```
permission denied                    (current host is not trusted by host auk)
```

Setting Up Server Hosts

Each build server host can have a bldserver.control file, which controls its use for parallel builds. This text file, /var/adm/Rational/config/bldserver.control, specifies when, how, and by whom the host can be used as a build server in a parallel build. This file can impose restrictions on who can use the host for parallel builds and when the host can be used for parallel builds. If a build server host has no such file, it accepts all parallel build requests.

During a parallel build, **clearmake** consults the user's build hosts file to determine which hosts to use for executing build scripts. Before actually dispatching a build script, **clearmake** queries the **albd_server** process on the target build host asking if **clearmake** can send a build script.

If the host's build server control file is missing or empty, no restrictions are placed on the use of the machine for parallel builds. The machine's **albd_server** always sends a **yes** response to the **clearmake** process that controls a parallel build.

If the host's build server control file is not empty, **albd_server** examines the load-balancing rules in order:

- If it finds a rule that matches the parameters of the current build, **albd_server** sends a **yes** response to the originating **clearmake**, which then uses a remote shell command to dispatch the build script.
- If no rule in the control file provides a match, **albd_server** sends a **no** response; the controlling **clearmake** proceeds to query another host.

For example, suppose this rule occurs in the control file:

```
-host jupiter -user *.dvt -time 21:00,07:30
```

This rule matches any build invoked on host **jupiter** between 9 P.M. and 7:30 A.M. by a user whose principal group is **dvt**.

To set up a build server host that is used for your team's daytime builds and its overnight builds:

1. Create a bldserver.control file. Each line of the bldserver.control file defines a situation in which it accepts parallel build requests.

```
% cat > /var/adm/Rational/config/bldserver.control
-time 08:30,19:30 -idle 60                    (1)
-time 19:30,05:30                             (2)
-user bldmeister                              (3)
<CTRL+D>
```

Line 1 specifies that during the interval between 8:30 A.M. and 7:30 P.M., this host accepts a parallel request when it is at least 60% idle. Line 2 specifies that during the interval between 7:30 P.M. and 5:30 A.M., this host accepts any parallel request, no matter how busy it is. Line 3 specifies that a parallel build request from a **clearmake** invoked by user **bldmeister** is always accepted.

2 Protect the bldserver.control file to make sure that your access-control settings cannot be deleted or altered:

```
% chmod 444 /var/adm/atria/config/bldserver.control
```

Each of the following specifications is optional. A missing specification implies no restriction. The specifications are logically ANDed to form a test against the parameters of the current build.

–host *host-list*

Specifies client hosts that are allowed or not allowed to use the current host for builds. *host-list* is a comma-separated list, and white space is allowed. Each item on the list is a host name, as listed by **uname(1)**. The asterisk (*) is a wildcard that matches all host names. To exclude a host, use the logical NOT operator (**!**) with any host argument except *.

For example:

`-host !sleepy,!crashy,neon`	*(matches host neon, explicitly excludes hosts sleepy and crashy, and implicitly excludes all other hosts)*
`-host !grumpy`	*(matches any host except grumpy)*

Note: Be sure to include the name of the current host, if the command to perform a parallel build may ever be entered here.

–user *user-list*

Specifies users who are allowed or not allowed to use this host for builds. *user-list* is a comma-separated list, and white space is allowed. Each item on the list specifies a user by name or by number, with a group qualifier or without. For example:

jones	User whose login name is **jones**
jones.dvt	User **jones**, but only if logged in with principal group **dvt**.
jones.*	Equivalent to specifying **jones** without any group qualifier.
566	User with user ID 566

To exclude a user, use the logical NOT operator (**!**) with any user argument or with the asterisk (*). For example:

`-user !george`	*(matches all users except george)*
`-user !darren,!jo,susan`	*(matches user susan, excludes users darren and jo, and implicitly excludes all other users)*
`-user !*`	*(excludes all users)*

–idle *percentage* [%]

Allows use of this host only when its idleness is at least *percentage*, which must be an integer between 0 and 100, inclusive. Idleness is negatively correlated with the host's load factor, as shown by **uptime(1)**; the approximate correspondence is this:

Load	Idle percentage
0.0	100
0.5	68
1.0	47
2.0	22
4.0	almost 0

Note: The idleness threshold can be specified with **–idle** settings on both the client and server. If there is a conflict, the overall principle is that the build server host controls its fate. For example:

- A **clearmake** process is searching for hosts that are at least 50% idle (the default). A build server that appears to qualify because it is 70% idle is not used if its bldserver.control file includes the line **–idle 75**.
- A bldserver.control file on a build server host permits access, because it contains the line **–idle 60** and the host is currently 75% idle. However, **clearmake** does not dispatch a build script to this host, because the build hosts file specifies a higher threshold: **–idle 80**.

–power *factor*

(Must be specified alone, on a separate line) During the computation of the host's idleness, divides *factor* into the *percentage* specified with **–idle** (or into the system default). Thus, these two specifications are equivalent: *factor* must be a nonnegative floating-point number.

–idle 60 **–idle 20**

–power 3

This option allows you to model a powerful host—perhaps a multiprocessor—that is more capable of accepting work at a given idleness level. You can use **–power 3.0** or **–power 2.5** for a three-processor build server host. You can also model a relatively weak host, by assigning it a power value less than 1.0.

If a build server control file includes multiple **–power** lines, only the last one takes effect.

–time *start-time*,*end-time* ...

Specifies one or more intervals during which the host is available as a build server. *start-time* and *end-time* must be specified in 24-hour format:

hh:*mm* (hh = 0–23 ; mm = 0–59)

An interval can span midnight; for example, **17:00,8:00** specifies the interval from 5 P.M. to 8 A.M. the following day.

Examples

- Allow builds by users **jackson** and **jones**, initiated from any host, if the host is at least 75% idle and the time is between 10 P.M. and 6 A.M.

  -host * -user jackson,jones -idle 75 -time 22:00,06:00

- Allow anyone to use this host for parallel builds between 7 P.M. and 7 A.M.

  -time 19:00,7:00

- Declare this host to be three times as powerful (able to handle parallel build requests) as a standard host.

  -power 3.0

Starting a Parallel Build

To start a parallel build:

1 Set the CCASE_HOST_TYPE variable. The value of this variable determines which build hosts file that **clearmake** looks for in your home directory:

Value	Build hosts file that clearmake looks for
`sun5`	.bldhost.sun5
`SUN5`	.bldhost.SUN5
`day`	.bldhost.day
`night`	.bldhost.night

2 Invoke **clearmake**. To enable parallel building, use the **–J** command-line option or set the CCASE_CONC environment variable. To specify a build hosts file, use the **–B** option. To have **clearmake** choose a build hosts file, do not use **–B**.

For example, to specify a build hosts file and start a build that builds up to five targets concurrently, use one of the following methods:

```
% clearmake –J 5 –B ~/.bldhost.day my_target      (command-line options)
```

```
% setenv CCASE_CONC 5                            (environment variable)
% clearmake –B ~/.bldhost.day my_target
```

```
% setenv CCASE_HOST_TYPE day                     (environment variable and
% clearmake –J 5 my_target                        command-line option)
```

```
% setenv CCASE_CONC 5                            (environment variables)
% setenv CCASE_HOST_TYPE day
% clearmake my_target
```

Note: If you specify **–J**, but do not set the CCASE_HOST_TYPE variable or specify a build hosts file with **–B**, **clearmake** builds run in parallel on the local host.

Setting CCASE_HOST_TYPE in a Shell Startup Script

In some parallel build environments, you may find it convenient to have your shell startup script set CCASE_HOST_TYPE. For example, your team may support an application on several architectures.

In this situation, you build the application for a particular architecture as follows:

1 Log in to a host of that architecture.

2 Set a view and go to the appropriate directory.

3 Enter a **clearmake –J** command to start a parallel build.

To implement such a scheme:

1 Use architecture-specific build hosts files. Give each build hosts file a file name extension that names a target architecture: .bldhost.hpux9, .bldhost.sunos5, and so on. Typically, each file lists hosts of one architecture only. For example, all SunOS 5 hosts are listed in .bldhost.sunos5.

2 Set CCASE_HOST_TYPE according to the local host's architecture. Include a routine in your shell startup file that determines the hardware/software architecture of the local host, and sets CCASE_HOST_TYPE to one of the file name extension strings: **hpux9**, **sunos5**, and so on. Here is a code fragment from a C shell startup script:

```
set ARCHSTRING = "`uname -s ; uname -r`"
switch ("$ARCHSTRING")
case "Solaris 5*":
setenv CCASE_HOST_TYPE solaris5
breaksw
case "HP-UX 9*":
setenv CCASE_HOST_TYPE hpux9
breaksw
  ...
```

Preventing Parallel Builds of Targets

When **clearmake** builds a makefile target, there may be side effects that you cannot address in a makefile. For example, one of your build tools may create temporary files that are not guaranteed to have unique names and then delete them at the end of its processing. When you use this tool serially, there are no problems. However, if you invoke it in multiple parallel builds in **clearmake**, the tool may create identical files and cause the builds to interfere with each other.

You can solve this problem by using the **.NOTPARALLEL** special makefile target. To disable parallel building for a makefile, use this target without any arguments. For example:

.NOTPARALLEL:

To prevent specific targets from being built in parallel with each other, specify them as a set of arguments. Note that parallel builds are prevented only within the set of targets. For example:

.NOTPARALLEL: %.a
.NOTPARALLEL: x.c y.c

In this example, **clearmake** does not build any .a file in parallel with any other .a file, and x is not built in parallel with y. However, **clearmake** can build .a files in parallel with x, y, or any other file.

Preventing Exponential Invocations of abe

If **clearmake** is invoked recursively during a parallel build, the result may be more invocations of **abe** than you want or than the build servers can handle. To prevent this situation, use the **.NOTPARALLEL** special makefile target for high-level invocations of **clearmake**.

Building Software for Multiple Platforms

10

This chapter addresses the challenge of using a single source tree to develop an application for a variety of hardware and software platforms. It discusses various approaches, contrasting their advantages and disadvantages. An extended example incorporates some of the approaches.

Issues in Multiple Platform Development

Several issues arise in an environment where developers create and maintain several platform-specific variants of an application:

- Different source code is required for different variants. Different UNIX operating systems may use different functions to implement the same task (for example, **strchr(3)** or **index(3)**). Likewise, it may be necessary to include different header files for different variants (for example, string.h and strings.h).
- Different variants and various platforms may have different requirements. The differences may involve such particulars as compiler locations, compiler options, and libraries.
- Builds for different variants must be kept separate. Because there is one source tree, care must be taken to ensure that the object modules and executables for one architecture are not confused with those for other architectures. For example, the link editor must not try to create an executable using an object module that was built for another architecture.

Additional issues must be addressed if Rational ClearCase does not run on one of the target platforms. For a discussion of one such issue, see Chapter 11, *Setting Up a Build on a Non-ClearCase Host*.

Handling Source Code Differences

We recommend that you use the same files (that is, the same versions of file elements) in all builds, for all platforms. You can usually achieve this goal by using the standard UNIX approach: conditional compilation using the C preprocessor, **cpp(1)**. For

example, if header file string.h is to be used for the architecture whose **cpp** symbol is **ARCH_A**, and header file strings.h is to be used for architecture **ARCH_B**, use this code:

```
#ifdef ARCH_A
#include <string.h>
#else
#ifdef ARCH_B
#include <strings.h>
#endif /* ARCH_B */
#endif /* ARCH_A */
```

If a file element cannot be compiled conditionally (for example, a bitmap image), the traditional solution is to put architecture-specific code in different elements (for example, panel.image.sparc versus panel.image.mc68k). This approach requires that build scripts be architecture specific, too.

With ClearCase, you have the option of splitting the element into branches. The **ARCH_A** variant can be developed on the element's **/main/arch_a** branch; edits and builds for that variant are developed in a view configured with this rule:

```
element * /main/arch_a/LATEST
```

Other variants are developed on similar branches, each using a different view, configured with a rule like the one above. In such a situation, the element's **main** branch may not be used at all.

We recommend that you use this branching strategy only when necessary, because of its disadvantages:

- Each time platform-independent code is changed on one of the branches, you must merge the change to the other branches.
- Developers must create a view for each architecture. In each view, only one variant of the application can be built.

If you can do so, organize your code into architecture-specific subdirectories or architecture-specific VOBs.

Handling Build Procedure Differences

Ideally, a single file (that is, a single version of a file element) drives all architecture-specific builds. One way to accomplish this is to revise makefiles as follows:

- Regularize build scripts
- Replace architecture-specific constructs (for example, /bin/cc) with make macro invocations (for example, **$(CC)**)

- Use the **clearmake** include directive to incorporate architecture-specific settings of the **make** macros. For more information about the **include** directive, see Chapter 9, *Setting Up a Parallel Build*.

For example, suppose that source file main.c is compiled differently for two different architectures:

```
main.o:
       /usr/ucb/cc -c -fsingle main.c

main.o:
       /usr/bin/cc -c main.c
```

To merge these build scripts, use the compiler pathname and options in make macros **CC** and **CFLAGS** and place an architecture-specific include line at the beginning of the makefile:

```
include /usr/project/make_macros/$(BLD_ARCH)_macros
 ..
main.o:
       $(CC) -c $(CFLAGS) main.c
```

The files in the **make_macros** directory then have these contents:

```
CC     = /usr/5bin/cc                 /usr/project/make_macros/sun4_macros
CFLAGS = -fsingle

CC     = /usr/bin/cc                  /usr/project/make_macros/irix5_macros
CFLAGS =
```

The make macro **BLD_ARCH** acts as a selector between these two files. The value of this macro can be placed in an environment variable by a shell startup script:

```
setenv BLD_ARCH `uname -s`
```

Alternatively, developers can specify the value at build time. For example:

```
clearmake main BLD_ARCH="HP-UX10"
```

Alternative Approach Using imake

Note: The **imake** utility is distributed with many UNIX variants and available for free from the MIT Consortium.

The **imake** utility provides an alternative to the method of using **make** macros described in the previous section. The **imake** methodology also involves architecture-specific make macros, but in a different way. **imake** generates an architecture-specific makefile by running **cpp** on an architecture-independent template file, typically named imakefile.

A typical imakefile contains a series of **cpp** macros, each of which expands to a build target line and its corresponding multiline build script. Typically, the expansion itself is architecture independent:

```
MakeObjectFromSrc(main)                 (macro in 'imakefile')

main.o: $(SRC)/main.c                   (expansion in actual makefile)

     $(CC) -c $(CFLAGS)$(SRC)/main.c
```

imake places architecture-specific make macro settings at the beginning of the generated makefile. For example:

```
SRC    = ..
CC     = /usr/5bin/cc
CFLAGS = -fsingle
RM     = rm -f
```

An idiosyncrasy of **imake** is that makefiles are derived objects, not source files. The architecture-independent template file (imakefile) is the source file and must be maintained as a ClearCase element.

Segregating the Derived Objects of Different Variants

It is essential to keep derived objects (object modules, executables) built for different architectures separate. This section describes two approaches, though others are possible.

Approach 1: Use Architecture-Specific Subdirectories

Each variant of an application can be built in its own subdirectory of the source directory. For example, if the source files for the executable **monet** are located in the directory /usr/monet/src, the variants can be built in subdirectories /usr/monet/src/sun4, /usr/monet/src/irix5, and so on. The simplest approach is to have the makefile create view-private subdirectories for this purpose. But if you want to use different derived object storage pools for the different variants, you must create the subdirectories as elements (**mkdir** command) and then adjust their storage pool assignments (**chpool** command).

Because the derived objects for the different variants are built at different pathnames (for example, /usr/monet/src/sun4/main.o), they are segregated by variant, and **clearmake** never winks in an object built for another architecture.

This approach has several advantages:

- All variants of the application can be built in a single view.
- You do not need to consider whether to suppress winkin for some or all targets.

- Because the derived objects for different variants have different pathnames, it is easier to organize multiple-architecture releases.

But this approach may require changes to build scripts: the binaries for a build are no longer in the source directory, but in a subdirectory. The build script in *Alternative Approach Using imake* on page 159 is structured for this situation:

```
main.o: $(SRC)/main.c
  .
  .
```

Approach 2: Use Different Views

Perform builds for different platforms in different views (**sun4_bld_vu**, **irix_bld_vu**, and so on). A team of developers working on the same variant can share a view or each can work in an architecture-specific view.

In most cases, the build script that creates a derived object differs for each variant, as described in *Handling Build Procedure Differences* on page 158. If so, **clearmake** prevents winkin of derived objects built for another architecture. You can force the build script to be architecture specific by including a well-chosen message or comment. For example, if **BLD_ARCH** is used as described in *Handling Build Procedure Differences* on page 158, you can include this message:

```
@echo "Building $@ for $(BLD_ARCH)"
```

The disadvantage of this approach is that when an element is checked out, you can build only one variant of the application. Because the checked-out version is visible only in one view, builds of other variants (which take place in other views) do not select the checked-out version. You must check in the element before building other variants.

Another disadvantage is the number of views that may be required. For instance, if seven developers want to maintain their own views in which to build four variants, 28 views are required.

Multiple Architecture Example

This section presents an example of multiple-architecture development. This example uses **imake** to support building in architecture-specific subdirectories.

Scenario

This section shows how to set up multiple-platform development in the /proj/monet/src directory.

You can perform a build for a particular architecture as follows:

1. Log on to a host of the desired architecture, for example, a workstation running Solaris 2.5.
2. In your regular view, move to the source directory, /proj/monet/src.
3. Enter the command **clearmake Makefiles** to have **imake** create the appropriate makefile in the architecture-specific subdirectory sun5. Note that the makefile is a derived object, not a source file. Thus, there is no need to create an element from this file.
4. Move to the sun5 subdirectory and build software for that architecture using **clearmake**.

The sections that follow describe how **imake** is involved in each of these steps.

Defining Architecture-Specific CPP Macros

Step 1 places you in an environment where the C preprocessor, **cpp**, defines one or more architecture-specific symbols. On a SunOS–4 host, **cpp** defines the symbols **sun** and **sparc**. This, in turn, causes **imake** to generate many architecture-specific (machine-dependent) **cpp** macros:

```
#ifdef sun                          sun defined by C preprocessor
#undef sun
#define SunArchitecture
#ifdef mc68020
...
#endif
#ifdef sparc                        sparc defined by C preprocessor
#undef SUN4
#undef sun4
#define MachineDep SUN4             imake defines longer symbols
#define machinedep sun4
#endif
...
```

Additional **cpp** macros specific to Sun are read in from the auxiliary file sun.cf.

Creating Makefiles in the Source and Build Directories

The **Imakefile** file in the source directory is the **imake** input file. This file controls the creation of makefiles in both the source directory itself and in the architecture-specific subdirectories where software is built:

```
#ifndef InMachineDepSubdir
  .
<code to generate makefile in source directory>
  .
#else
  .
<code to generate makefile in an architecture-specific subdirectory>
  .
#endif
```

The **Imakefile** code used in the source directory defines a symbol to record the fact that builds do not take place in this directory:

```
#define IHaveMachineDepSubdirs
```

The **Makefile** that **imake** generates includes a Makefiles target that populates an architecture-specific subdirectory with its own makefile. The CPU environment variable determines the name of the architecture-specific subdirectory.

```
Makefiles::
    @echo "Making Makefiles in $(CURRENT_DIR)/$$CPU"
    -@if [ ! -d $$CPU ]; then \
        mkdir $$CPU; \
        chmod g+w $$CPU; \
    else exit 0; fi
    @$(IMAKE_CMD) -s $$CPU/Makefile \
    -DInMachineDepSubdir \
    -DTOPDIR=$(TOP) -DCURDIR=$(CURRENT_DIR)/$$CPU
```

The command **clearmake Makefiles** invokes **imake** again, using the same **Imakefile** for input. This time, the symbol **InMachineDepSubdir** is defined, which causes the actual build code to be generated.

The **Imakefile** in /proj/monet/src contains these macros:

```
OBJS = cmd.o main.o opt.o prs.o
LOCAL_LIBRARIES = ../../lib/libpub/libpub.a

MakeObjectFromSrc(cmd)
MakeObjectFromSrc(main)
MakeObjectFromSrc(opt)
MakeObjectFromSrc(prs)

ComplexProgramTarget(monet)
```

The makefile generated in the build directory, /proj/monet/src/sun4, includes this build script:

```
$(AOUT): $(OBJS) $(LOCAL_LIBRARIES)
    @echo "linking $@"
    -@if [ ! -w $@ ]; then $(RM) $@; else exit 0; fi
    $(CC) -o $@ $(OBJS) $(LOCAL_LIBRARIES) \
        $(LDFLAGS) $(EXTRA_LOAD_FLAGS)
```

Setting Up a Build on a Non-ClearCase Host

11

This chapter describes a technique for creating configuration records for a build that involves ClearCase data, but is performed on a non-ClearCase host. *Non-ClearCase access* (exporting a VOB through a view) makes the data available to that host; a remote shell is invoked to perform the build on that host.

Build Scenario

Suppose you want to build library libpub.a for an architecture that Rational ClearCase does not currently support, using a host of that architecture named **titan**. The VOB storage area for the library's sources is located at /vobstore/libpub.vbs on host **sol**. This VOB is also mounted on **sol**, at /proj/libpub.

Setting Up an Export View

A ClearCase export view allows limited access to one or more VOBs by using standard NFS export facilities. Each NFS export provides remote access to one VOB through a particular view. The following limitations apply when you use an export view to access a VOB:

- You cannot check out or check in versions in the VOB. If you need to check out a file, you must perform a remote login to a ClearCase host.
- Builds in the view do not perform build auditing, configuration lookup, or winkin. The builds do not create derived objects or config records. Any files created during the build are view-private objects.

  Note: You can use remote-shell techniques to overcome this limitation, so that the files built on a non-ClearCase host become derived objects. See *Revising the Build Script*.
- You cannot reconfigure the view from the non-ClearCase host. If you need to reconfigure the view, you must perform a remote login to a ClearCase host.

  Note: If you modify an export view's config spec, all users who may currently have the view mounted for non-ClearCase access must unmount and remount the view.

Remounting the view ensures access to the correct set of files as specified in the updated config spec.

For information about setting up views and VOBs for export, see the *Administrator's Guide* for Rational ClearCase.

Note: Export views are to be used only for non-ClearCase access to VOBs. To make a view accessible on a remote host, use the **startview** or **setview** command on that host. An export view can be mounted on a ClearCase host, but never try to mount it on the *dynamic-views root directory*, /view.

Mounting the VOB Through the Export View

On the non-ClearCase host, a standard NFS mount is performed on the exported pathname. For example, mount /view/libpub_expvu/proj/libpub at /proj/libpub (the same location at which the VOB is mounted on ClearCase hosts).

Revising the Build Script

To produce an audited build on a non-ClearCase host, you must revise the build script. Thus, it makes sense to build in an architecture-specific subdirectory, with a customized makefile. (For more information, see Chapter 10, *Building Software for Multiple Platforms*.)

To create a CR that lists all of the build's input files and output files, the build script executed by **clearmake** must do the following:

- Declare all input files as explicit dependencies. Because the MVFS does not run on the non-ClearCase host, source dependencies are not detected.
- Invoke a remote shell to perform the build on the non-ClearCase host.
- If the build performed by the remote shell succeeds, run the **touch(1)** on all output files from the ClearCase host. This command converts the view-private files created by the remote shell command to derived objects.

A simple build script can be transformed as follows:

Native Build

```
OBJS = data.o errmsg.o getcwd.o lineseq.o

data.o:                                    (source dependencies need not be
                                           declared)
```

Native Build

```
 cc -c data.c
.                                                    (other object modules produced
.                                                    similarly)
.
libpub.a: $(OBJS)
 ar -rc $@ $(OBJS)
```

Non-ClearCase Build

```
OBJS = data.o errmsg.o getcwd.o lineseq.o


data.o: data.c libpub.h                              (must declare source dependencies)


 rm -f $@
 rsh titan 'cd /proj/libpub ; cc -c
data.c'
 if [ -w $@ ]; then \
 touch $@ ; \
 fi
.                                                    (other object modules produced
.                                                    similarly)
.
libpub.a: $(OBJS)
 rm -f $@
 rsh titan 'cd /proj/libpub ; ar -rc $@
$(OBJS)'
 if [ -w $@ ]; then \
 touch $@ ; \
 fi
```

The remote shell command (**rsh** in the example above) varies from system to system.

The remote shell program typically exits with a status of zero, even if the compilation fails. Thus, you must use some other technique to verify the success of the build after the remote shell returns. In this example, the build scripts assume that the remote build is successful if the target file exists and is writable.

Performing an Audited Build in the Export View

To perform the desired build:

1. Register and set the export view on your workstation, which is a ClearCase host:

   cleartool mktag –tag libpub_expvu /public/export.vws
   cleartool setview libpub_expvu

2. Build in the normal way, on your host:

   cd /proj/libpub
   clearmake

The script listed above specifies a particular non-ClearCase host, **titan**, on which remote shells are to be executed. If builds are performed on more than one non-ClearCase host, you must generalize this script.

Note: Because the remote host name is part of the build script, winkin of derived objects built on the various hosts fails unless you make further modifications (for example, by using **clearmake –O** to disable build-script checking).

Index

.class files 133–134
.class.dep files 133
  derived 135
  format 136
  using 136
.cmake.state file 19
.dep files 133
  derived 135
  format 136
  using 136
.DEPENDENCY_IGNORED_FOR_REUSE 140
.JAVA_TGTS target 79
.JAVAC
  building applications without 137
  in BOS files 137
  makefile requirements 133
  makefile target 132
  unsupported with nonaudited or parallel builds 133
.JAVAC target 80
.MAKEFILES_AFFECT_REUSE target 81
.MAKEFILES_IN_CONFIG_REC target 80
.NOTPARALLEL target 156

A

abe (audited build executor) 141–142, 156
abe process, use for parallel builds 141
ALL_CLASSES
  using with javaclasses macro 134
ar, use of u key 45
archives
  format in makefile 72
  incremental update example 42
attributes, attaching to versions in CR 66

B

bldserver.control file 150
BOS files
  about 70
  clearmake read order 70
  format of contents 83
  recommended use 70
  special targets for 78
branch strategy for multiple platforms 158
build auditing
  about 4
  effect of background processes 41
  in export views 168
  including non-MVFS files 34
  incremental updates and 42
  multiple levels, problems 40
  without clearmake 8
build avoidance
  about 5
  algorithms for make and clearmake 43
  Cfront-based compilers 94
  differences in clearmake and make 30
  multiple build scripts for target 31
  scheme for in make 33
  template instantiation method 90
build environment
  for clearmake and make 27
  views used 1
build hosts

client setup for parallel builds 143
non-ClearCase 10
non-ClearCase, setting up 165
parallel builds 149
server setup 150
build hosts files 144
build scheme in ClearCase 1
build scripts
DO-IDs in 53
format in makefile 69
multiple for single target 31
non-ClearCase hosts 166
parallel builds 141
temporary changes to 30
when omitted from CRs 16
builds
DOs and performance 7
forced, problems with 36
how they work 3
labeling versions created in 40
reference time 15
reference time and build sessions 3
starting 27
subsessions 40
verbosity levels, increasing 30
working while in progress 37
builds for multiple platforms
branching strategy 158
DOs and subdirectories 160
DOs and views 161
example with imake 161
handling source code differences 157
makefiles for 158
built-in rules in makefiles 73

C

C++ templates
about 90
alternative instantiation procedures 91
recommended method of instantiation 90
catcr command
DO versions 58
sample listing 13
CCASE_BLD_UMASK environment variable 49
CCASE_HOST_TYPE environment variable 143, 145, 153–154
CCASE_OPTS_SPECS environment variable 71
CCASE_VERBOSITY environment variable 30
ccase-home-dir directory xvii
Cfront-based compilers
about 91
Forced Instantiation model 104
how link-time template instantiation works 93
interaction with clearmake 92
Multiple Repositories model 98
Simple instantiation model 95
clearaudit
about 8
contents omitted from CR 16
coordinating multiple builds 40
multiple log files, workarounds 40
use with make programs 87
clearmake
build scenario 27
compatibility modes 8, 87
declaring dependencies in makefiles 33
double-colon rules 36
format of makefiles 69
increasing verbosity level for builds 30
interaction with SGI Delta/C++ compiler 120

interaction with XLC compiler 122
interactions with Cfront-based compilers 92
interactions with SPARCompiler C++ 4.x 108
internal macros 75
invoking 27
link-time template instantiation problems 94
macro substitution 74
recursive invocation 32
standard input as makefile 72
symptoms of template problems for C++ 90
clock skew
about 38
time rules and 39
commands, control of echoing during build 72
compatibility modes in clearmake
about 87
adjusting levels of 8
config specs, time rules in 38
configuration lookup
about 5
common outcomes 6
in hierarchical builds 7
problems with dependencies 35
VPATH macro 76
conventions, typographical xvii
cquest-home-dir directory xvii
CRs (configuration records)
about 5
attaching labels and attributes to versions in 66
cache 19
comparing 66
contents of 13
contents of, effect of background processes 41
displaying contents of 66
displaying for DO versions 58
double-colon rules and contents of 36
effect on DOs when unavailable 65
example 13
hierarchy of 16
hierarchy, and winkin 54
hierarchy, processing by cleartool commands 19
how created 41
incremental updates and 42
merging dependencies in 44
recording makefile version 67
storage of 19
customer support xix

D

.DEFAULT target 77
dependencies
build order, in makefiles 36
declaring in makefiles 33
detected, log of 7
format in makefiles 69
merging in CRs 44
problems when searching directories for 35
tracking 4
tracking non-MVFS files 5
.DEPENDENCY_IGNORED_FOR_REUSE target 78
describe command 51
diffcr command 66
DO versions
about 24
access to 58
as release mechanism 61
creating 57

creating in builds 57
displaying configuration records 58
displaying description of 51
documentation
Help description xviii
DO-IDs
about 12
displaying 51
in build scripts 53
in cleartool commands 53
vs. OID 51
DOs (derived objects)
about 5, 11
attaching labels and attributes to sources in CR 66
build avoidance role 5
converting to view-private files 56
costs of creating 7
criteria for reuse or winkin 6
degenerate 65
disk space usage, displaying 64
displaying kind of 50
effect of forced builds 36
incremental updating 42
incremental updating by Cfront-based compilers 94
incremental updating, example 42
incremental updating, links in 45
incremental updating, scenarios 45
kinds of 20
listing at specific pathnames 50
listing views that reference 52
multiple platform, separating 160
overwriting 64
removing 64
scrubbing 64
selecting versions for in view 66
siblings of 5
siblings of, types 21
specifying in commands 53
storage 21
when created 29
dospace command 64
double-colon rules, how clearmake interprets 36

E

environment variables 72
CCASE_BLD_UMASK 49
CCASE_HOST_TYPE 153
CCASE_OPTS_SPECS 71
CCASE_VERBOSITY 30
order of precedence in makefiles 73
SHELL 86
error handling, control of in makefile 72
exit status 4
export views
about 165
auditing builds in 168
mounting VOBs 166
express builds
about 7
creating views for 56
reconfiguring views for 55
when to use 55
winkin to 55

F

function names in makefiles 72

H

hard links
  VOB, creating to DOs 63
  when winked in 25
header files, precompiled 91
Help, accessing xviii
hierarchical builds
  configuration lookup in 7
  reference time of 15
  use of 40
HP aC++ compiler
  about 127
  template instantiation models 127

I

idleness threshold 147
.IGNORE target 77
imake utility
  about 159
  build example 161
include file facility 148
include files in makefiles 73
including comments in a file 148
.INCREMENTAL_REPOSITORY_SIBLING target 78
.INCREMENTAL_TARGET target 79

J

Java class dependencies
  cycle 136
  deriving 135
  DO winkin and reuse 136
  nested classes 137
  storing 136
Java compilers
  configuring makefiles 140
  makefiles for 138
  rebuilding targets 139
  using make with 131
javac
  ClearCase build problems 132
  using with clearmake 131
  using with make 131
javaclasses
  built-in macro 134

L

labels, attaching to versions in CR 66
libraries, format in makefile 72
load balancing 146
lsdo command 52
  examples 50
lsprivate command 50

M

macros
  internal clearmake 75
  order of precedence in makefiles 73
  substitution by clearmake 74
  target-dependent definitions 70
$(MAKE) macro, defining for clearmake 32
make
  about 1
  build avoidance scheme 33
  use with Java 131
make macros
  format in makefile 70

format of definition 74
temporary overrides of 30
MAKEARGS 142
makefiles
about 69
builds for multiple platforms 158
built-in rules 73
controlling execution of 72
declaring dependencies in 33
design models for Cfront-based compilers 95
double-colon rules and clearmake 36
format for clearmake 69
format of libraries 72
function names in 72
imakefiles 159
include files in 73
incremental updating and 43
Java compilers 138
javac, using with 131
non-MVFS dependencies and 5
order of precedence, macros and environment variables 73
overriding build scripts in 30
special targets 77
standard input as, in clearmake 72
UNIX, on Windows NT 83
version of in CR 67
MAKEFLAGS 142
MFLAGS 142
mounting VOBs, in export views 166
MVFS
template instantiation outside 91
MVFS files
about 4
in configuration records 16

N

.NO_CMP_NON_MF_DEPS target 81
.NO_CMP_SCRIPT target 81
.NO_CONFIG_REC target 81
.NO_DO_FOR_SIBLING target 81
.NO_WINK_IN target 82
non-ClearCase hosts
about 10
setup for build 165
non-MVFS files
as dependencies, tracking 5
in configuration records 16
nonshareable DOs
about 7, 20
automatic conversion to shareable 62
converting to shareable 62
promotion and winkin 21
storage 21
types of siblings 21
unique DO-IDs for 24
.NOTPARALLEL target 82
uses of 155

O

OIDs
how used 15
order of precedence in makefiles 73

P

parallel builds
about 9, 141
client setup 143
how clearmake works 9

how controlled 141
preventing parallel, of targets 155
scheduler 142
server setup 150
starting 153
pathnames
accessing DOs 53
and DO-IDs 12
for DO versions 58
form in build scripts 69
.PRECIOUS target 77
pseudotargets, and winkin 32

R

reference count
about 25
hard links and 25
when zero 64
reference time
about 15
effect on source control 37
for multiple builds 40
release areas, structure and management 61
rmdo command 64

S

scrubbing DOs 64
SGI Delta/C++ compiler 4.x
about 119
interactions with clearmake 120
template instantiation models 120
shareable DOs
about 20
components of 21
converting to nonshareable 62
in views reconfigured for express builds 56
permissions to share 49
promotion and winkin 21
removing data containers 64
storage 21
types of siblings 21
unique DO-IDs for 24
shell
auditing build in 8
SHELL environment variable 86
shells
setting CCASE_HOST_TYPE in 154
.SIBLING_IGNORED_FOR_REUSE target 82
siblings of DOs
about 5
shareable and nonshareable 21
.SIBLINGS_AFFECT_REUSE target 83
.SILENT target 77
SPARCompiler C++ 4.x
about 108
interaction with clearmake 108
makefile example 115
Multiple Repositories model 113
repository cleanup 109
repository setup 109
Simple model 110
subsessions in builds 40
subtargets in makefiles 36
symbolic links 11

T

targets
build rules and clearmake macros 75

format in makefiles 69
multiple build scripts for 31
preventing parallel builds 155
rebuilding by Java compilers 139
recursive invocation of clearmake 32
special 77
special, format in makefile 70
special, lists of 77
time rules
effect of clock skew 39
use in config specs 38
time stamps, adding to C-language executables 46
trust relationships 149
typographical conventions xvii

U

umask setting to share DOs 49

V

version strings, adding to C-language executables 46
versions
checked-out, how clearmake handles 29
created in builds, labeling 40
of DOs 24
view-extended pathnames for DOs 53
views
configuring for express builds 55
configuring to select versions for DO 66
context 3
for builds 1
preventing winkin to and from 54
references to DOs, listing 52
time rules and 38
using in builds for multiple platforms 161
VPATH macro 76

W

what string, creating 46
winkin
about 6
criteria for 6
from other platforms, preventing 56
hard links and 25
manual 54
permissions for 50
preventing 54
pseudotargets and 32
recursive, uses of 54
reference count and 25

X

XLC compiler
about 122
Compile-Time Demand Instantiation model 125
Explicit Instantiation model 126
interaction with clearmake 122
Simple model 123