C++ Migration Guide

901 San Antonio Road
Palo Alto, , CA 94303-4900
USA 650 960-1300 fax 650 969-9131

Part No: 805-4956
Revision A, February 1999

Copyright Copyright 1999 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

All rights reserved. This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® system, licensed from Novell, Inc., and from the Berkeley 4.3 BSD system, licensed from the University of California. UNIX is a registered trademark in the United States and in other countries and is exclusively licensed by X/Open Company Ltd. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers. RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

Sun, Sun Microsystems, the Sun logo, SunDocs, SunExpress, Solaris, Sun Performance Library, Sun Performance WorkShop, Sun Performance WorkShop Fortran, Sun Visual WorkShop, Sun WorkShop, Sun WorkShop Compilers C, Sun WorkShop Compilers C++, Sun WorkShop Compilers Fortran, Sun WorkShop Memory Monitor, Sun WorkShop Professional, Sun WorkShop Professional C, and Sun WorkShop TeamWare are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox Corporation in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a nonexclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Copyright 1999 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, Californie 94303-4900 U.S.A. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Des parties de ce produit pourront être derivées du système UNIX® licencié par Novell, Inc. et du système Berkeley 4.3 BSD licencié par l'Université de Californie. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays, et licenciée exclusivement par X/Open Company Ltd. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, the Sun logo, SunDocs, SunExpress, Solaris, Sun Performance Library, Sun Performance WorkShop, Sun Performance WorkShop Fortran, Sun Visual WorkShop, Sun WorkShop, Sun WorkShop Compilers C, Sun WorkShop Compilers C++, Sun WorkShop Compilers Fortran, Sun WorkShop Memory Monitor, Sun WorkShop Professional, Sun WorkShop Professional C, et Sun WorkShop TeamWare sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

Les utilisateurs d'interfaces graphiques OPEN LOOK® et Sun™ ont été développés de Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox Corporation pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique, cette licence couvrant aussi les licenciés de Sun qui mettent en place les utilisateurs d'interfaces graphiques OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A REPONDRE A UNE UTILISATION PARTICULIERE OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.

Adobe PostScript

Please Recycle

Contents

Preface

This book explains what you need to know when moving from 4.0, 4.0.1, 4.1, or 4.2 versions of the C++ compiler to the C++ 5.0 version. If you are moving from still earlier 3.0 or 3.0.1 versions of the C++ compiler, the information still applies. A few additional topics specific to these older compiler versions are addressed.

Who Should Use This Book

This manual is intended for programmers with a working knowledge of C++ and some understanding of the Solaris™ operating environment and UNIX® commands.

How This Book Is Organized

This book contains the following chapters:

Chapter 1, "Introduction," discusses the C++ language changes, the compiler compatibility modes, binary compatibility issues, conditional expressions, and function pointers and `void*`.

Chapter 2, "Using Compatibility Mode," describes the minor differences between using the C++ 4.0, 4.1, or 4.2 compiler and using the C++ 5.0 compiler.

Chapter 3, "Using Standard Mode," explains the C++ 5.0 compiler changes.

Chapter 4, "Using Libraries and Header Files," explains library and header file changes in the C++ 5.0 compiler.

Chapter 5, "Migrating from C++ 3.0 to C++ 5.0," discusses migration from the C++ 3.0 compilers to the C++ 5.0 compiler.

Chapter 6, "Moving from C to C++," describes how to move programs from C to C++.

Multiplatform Release

The Sun™ WorkShop™ C++ compiler documentation applies to the release of the C++ compiler on Solaris 2.5.1, 2.6, and Solaris 7 operating environments on:

- The SPARC™ platform
- The x86 platform, where x86 refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent

---

Note - The latest operating environment release is Solaris 7, but code and path or package path names may use Solaris 2.7 or SunOS 5.7.

Note - The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms "SPARC" and "x86" in the text.

C++ Books

The following books are part of the C++ 5.0 documentation package.

- *C++ User's Guide* instructs you in the use of the C++ 5.0 compiler and provides detailed information on command-line options.
- *C++ Programming Guide* tells you how to use C++ 5.0 features to write more efficient programs. Some of the areas discussed include using templates, exception handling, casting, runtime type identification, and interfacing with FORTRAN 77.
- C++ Library Reference describes the C++ libraries, including the C++ Standard Library, the `Tools.h++` Class Library, the Sun WorkShop Memory Monitor, and the iostream and complex libraries.
- *Tools.h++ User's Guide* discusses use of the C++ classes for enhancing the efficiency of your programs.
- *Tools.h++ Class Library Reference* provides details on the `Tools.h++` class library.

- *C++ Standard Library 2.0 User's Guide* instructs you in the use of the C++ Standard Library, including locales and iostreams.
- *C++ Standard Library Class Reference* provides more detailed information on the use of the C++ Standard Library.
- *Sun WorkShop Memory Monitor User's Guide* describes how to use the Sun WorkShop Memory Monitor garbage collection and memory management tools.

Other Sun WorkShop Books

The following books are part of the Sun Visual WorkShop C++ documentation package:

- *Sun WorkShop Quick Install* provides installation instructions.
- *Sun WorkShop Installation and Licensing Reference* provides supporting installation and licensing information.
- *Sun Visual WorkShop C++ Overview* gives a high-level outline of the C++ package suite.
- *Using Sun WorkShop* gives information on performing development operations through Sun WorkShop.
- *C User's Guide* tells how to use the C compiler.
- *Numerical Computation Guide* details floating-point computation numerical accuracy issues.
- *Debugging a Program With* `dbx` provides information on using `dbx` commands to debug a program.
- *Analyzing Program Performance With Sun WorkShop* describes the profiling tools; the LoopTool, LoopReport, and LockLint utilities; and use of the Sampling Analyzer to enhance program performance.
- *Sun WorkShop TeamWare User's Guide* describes how to use the Sun WorkShop TeamWare code management tools.
- *Sun WorkShop Performance Library Reference Manual* discusses the library of subroutines and functions to perform useful operations in computational linear algebra and Fourier transforms.
- *Sun WorkShop Visual User's Guide* describes how to use Visual to create C++ and Java™ graphical user interfaces.

Solaris Books

The following Solaris manuals and guides provide additional useful information:

- *The Solaris Linker and Libraries Guide* gives information on linking and libraries.

- The Solaris *Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS™ system.

Commercially Available Books

The following is a partial list of available books on the C++ language.

Object-Oriented Analysis and Design with Applications, Second Edition, Grady Booch (Addison-Wesley, 1994)

Thinking in C++, Bruce Eckel (Prentice Hall, 1995)

The Annotated C++ Reference Manual, Margaret A. Ellis and Bjarne Stroustrup (Addison-Wesley, 1990)

Design Patterns: Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, (Addison-Wesley, 1995)

C++ Primer, Third Edition, Stanley B. Lippman and Josee Lajoie (Addison-Wesley, 1998)

Effective C++-50 Ways to Improve Your Programs and Designs, Second Edition, Scott Meyers (Addison-Wesley, 1998)

More Effective C++-35 Ways to Improve Your Programs and Designs, Scott Meyers (Addison-Wesley, 1996)

STL Tutorial and Reference Guide-Programming with the Standard Template Library, David R. Musser and Atul Saini (Addison-Wesley, 1996)

C++ for C Programmers, Ira Pohl (Benjamin/Cummings, 1989)

The C++ Programming Language, Third Edition, Bjarne Stroustrup (Addison-Wesley, 1997)

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at `http://www.sun.com/sunexpress`.

Accessing Sun Documents Online

Online documentation includes: online books, man pages, command-line help and README files. These are described below.

Online Books

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject. The URL is `http://docs.sun.com/`.

The Sun WorkShop documentation is also available using AnswerBook™ software. To access the AnswerBook collections, your system administrator must have installed the AnswerBook package during the installation process. For information on installing and accessing AnswerBook software see the Sun WorkShop installation documentation, the Solaris installation documentation, or your system administrator.

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help Release Notes.

Man Pages

Online man pages provide immediate documentation about a command or library function. You can display a man page by running the command:

```
demo% man topic
```

Man pages are in:

opt-install-dir/`SUNWspro/man`

The following table lists and describes the C++ man pages.

Note - Before you use the `man` command, at the beginning of your search path, insert the name of the directory in which you have chosen to install the C++ compiler. This enables you to use the `man` command. This is usually done in the `.cshrc` file, in a line with `setenv MANPATH` at the start; or in the `.profile` file, in a line with `export MANPATH` at the start.

TABLE P–1 C++ Man Pages

Title	Description
CC	Describes the C++ compiler command line options
`CCadmin`	Cleans the templates database. Provides information about mangled names and symbols in the templates database
`cartpol`	Provides Cartesian/polar functions in the C++ complex number math library
`cplx.intro`	Introduces the C++ complex number math library
`cplxerr`	Provides complex error-handling functions in the C++ complex number math library
`cplxops`	Provides arithmetic operator functions in the C++ complex number math library
`cplextrig`	Provides trigonometric operator functions in the C++ complex number math library
`demangle`	Decodes a C++ encoded symbol name
`filebuf`	Buffer class for file I/O
`fstream`	Provides stream class for file I/O
`istream`	Supports formatted and unformatted input
`ios`	Provides basic iostream formatting
`ios.intro`	Introduces iostream man pages
`manip`	Provides iostream manipulators

TABLE P–1 C++ Man Pages *(continued)*

Title	Description
`ostream`	Supports formatted and unformatted output
`queue`	Provides list management for task library
`sbufprot`	Provides protected interface of streambuffer base class
`sbufpub`	Provides public interface of streambuffer base class
`sigfpe`	Allows signal handling for specific SIGFPE codes
`ssbuf`	Provides buffer class for character arrays
`stdarg`	Handles variable argument list
`stdiobuf`	Provides buffer and stream classes for use with C stdio
`stream_locker`	Provides class used for application level locking of iostream class object
`stream_MT`	Base class that provides dynamic changing of iostream class object to and from MT safely
`strstream`	Provides stream class for "I/O" using character arrays
`varargs`	Handles variable argument list
`vector`	Provides generic vector and stack

The following table lists man pages that contain information related to the C++ compiler.

TABLE P–2 C++-Related Man Pages

Title	Description
`c++filt`	Copies each file name in sequence and writes it in the standard output after decoding symbols that look like C++ demangled names.
`dem`	Demangles one or more C++ names that you specify
`fbe`	Creates object files from assembly language source files.
`fpversion`	Prints information about the system CPU and FPU
`gprof`	Produces execution profile of a program
`ild`	Links incrementally, allowing insertion of modified object code into a previously built executable
`inline`	Expands assembler inline procedure calls
`lex`	Generates lexical analysis programs
`rpcgen`	Generates C/C++ code to implement an RPC protocol
`version`	Displays version identification of object file or binary
`yacc`	Converts a context-free grammar into a set of tables for a simple automaton that executes an LALR(1) parsing algorithm

`README` file

The `README` file highlights important information about the compiler, including:

- New/changed features
- Software incompatibilities
- Current software bugs
- Information discovered after the manuals were printed

README files are in:

opt-install-dir`/SUNWspro/READMEs`

View the C++ `README` file by typing

```
%CC -readme
```

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

TABLE P–3 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`**
		`Password:`

AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Compiler options and code samples may use the following conventions:		
`[ ]`	Square brackets contain arguments that are optional.	`-xO[n]`
( )	Parentheses contain a set of choices for a required option.	–d(y | n)
|	The “pipe” or “bar” symbol separates arguments, only one of which may be used at one time.	–d(y | n)
`...`	The ellipsis indicates omission in a series.	`-xinline=`*f1*[,...*fn*]

TABLE P–3 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
`%`	The percent sign indicates the word has a special meaning.	`-ftrap=%all, no%division`
`<>`	In ASCII files, such as the `README` file, angle brackets contain a variable that must be replaced by an appropriate value.	`-xtemp=<`*dir*`>`

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–4 System Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

CHAPTER 1

Introduction

In this book, the C++ 4.0, 4.0.1, 4.1, and 4.2 compilers are referred to collectively as "C++ 4." To a large degree, C++ source code that compiled and ran under C++ 4 continues to work under the C++ 5.0 compiler, with a few exceptions that are due to changes in the C++ language definition. The C++ 5.0 compiler provides a compatibility mode (`-compat=4`) that allows nearly all of your C++ 4 code to continue to work unchanged.

Note - C++ 5.0 object code compiled in standard mode is not compatible with C++ code from any earlier compiler. It is still possible to use self-contained libraries of older object code with the C++ 5.0 compiler. The details are covered in "Binary Compatibility Issues" on page 3.

The C++ Language

C++ was first described in *The C++ Programming Language* (1986) by Bjarne Stroustrup, and later more formally in The Annotated C++ Reference Manual (the ARM) (1990), by Margaret Ellis and Bjarne Stroustrup. The Sun C++ 4 compiler versions were based primarily on the definition in the ARM, with additions from the then-emerging C++ standard. The additions selected for inclusion in C++ 4, and particularly in the C++ 4.2 compiler, were mainly those that did not cause source and binary incompatibility.

C++ is now the subject of an international standard, ISO/IEC 14882:1998 *Programming Languages - C++*. The C++ 5.0 compiler in standard mode implements nearly all of the language as specified in the standard. The README file that accompanies the current release describes any departures from requirements in the standard.

Some changes in the C++ language definition prevent compilation of old source code without minor changes. The most obvious example is that the entire C++ standard library is defined in namespace `std`. The traditional first C++ program

```
#include <iostream.h>
int main() { cout << ‘‘Hello, world!’’ << endl; }
```

no longer compiles under a strictly-conforming compiler because the standard name of the header is now `<iostream>` (without the `.h`), and the names `cout` and `endl` are in namespace `std`, not in the global namespace. The C++ 5.0 compiler, as an extension, provides a header `<iostream.h>` that allows that program to compile even in standard mode. Besides the source code changes required, such language changes create binary incompatibilities, and so were not introduced into the Sun C++ compiler prior to version 5.0.

Some newer C++ language features also required changes in the binary representation of programs. This subject is discussed in some detail in “Binary Compatibility Issues” on page 3.

Compiler Modes of Operation

The C++ 5.0 compiler has two modes of operation, *standard* mode and *compatibility* mode.

Standard Mode

Standard mode implements most of the C++ International Standard, and has some source incompatibilities with the language accepted by C++ 4, as noted earlier.

More importantly, the C++ 5.0 compiler in standard mode uses an Application Binary Interface (ABI) different from that of C++ 4. Code generated by the compiler in standard mode is generally incompatible with, and cannot be linked with, code from the various C++ 4 compilers. This subject is discussed in more detail in “Binary Compatibility Issues” on page 3.

You should update your code to compile in 5.0 standard mode, for several reasons:

- Compatibility mode is not available for 64-bit programs.
- You can’t use important standard C++ features in compatibility mode.
- New code written to the C++ standard might not compile in compatibility mode, meaning you can’t import future new code into the application.

- Since you can’t link 4.2 and standard-mode C++ code together, you might need to maintain two versions of object libraries.
- Compatibility mode will not be supported forever.

Compatibility Mode

To provide a migration path from C++ 4 to C++ 5.0 standard mode, the C++ 5.0 compiler provides a compatibility mode. The compatibility mode is fully binary compatible and mostly source compatible with the C++ 4 compiler. (*Compatible* means *upward compatible*. Older source and binary code works with the new compiler, but you cannot depend on code intended for the new compiler working with an old compiler.) Compatibility mode is not binary compatible with standard mode. Compatibility mode is available for Intel and SPARC on Solaris 2.5.1, 2.6, and Solaris 7, but not for SPARC V9 (64-bit Solaris 7).

Reasons to use compatibility mode:

- You have C++ object libraries compiled with a 4.2 compiler and you can’t recompile them in 5.0 standard mode. (For example, you don’t have the source code.)
- You need to get a product out the door immediately, and your source code won’t compile in 5.0 standard mode.

Binary Compatibility Issues

An *Application Binary Interface*, or ABI, defines the machine-level characteristics of the object program produced by a compiler. It includes the sizes and alignment requirements of basic types, the layout of structured or aggregate types, the way in which functions are called, the actual names of entities defined in a program, and many other features. Much of the C++ ABI for Solaris is the same as the basic Solaris ABI, which is the ABI for the C language.

Language Changes

C++ introduced many features (such as class member functions, overloaded functions and operators, type-safe linkage, exceptions, and templates) which did not correspond to anything in the ABI for C. Each major new version of C++ added language features that could not be implemented using the previous ABI. Necessary ABI changes have involved the way class objects are laid out, or the way in which some functions are called, and the way type-safe linkage (“name mangling”) can be implemented.

The C++ 4.0 compiler implemented the language defined by the ARM. By the time the C++ 4.2 compiler was released, the C++ committee had introduced many new language features, some requiring a change in the ABI. Because it was certain that additional ABI changes would be required for as-yet unknown language additions or changes, Sun elected to implement only those new features that did not require a change to the ABI. The intent was to minimize the inconvenience of having to maintain correspondence of binary files compiled with different compiler versions. Now that the C++ standard has been published, Sun has designed a new ABI that allows the full C++ language to be implemented. The C++ 5.0 compiler uses the new ABI by default.

One example of language changes affecting the ABI is the new names, signatures, and semantics of the `new` and `delete` free-store functions. Another is the new rule that a template function and non-template function with the same signature are nevertheless different functions. This rule required a change in "name mangling," that created a binary incompatibility with older compiled code. The introduction of type `bool` also created an ABI change, particularly regarding the interface to the standard library. Because the ABI needed to change, aspects of the old ABI that resulted in needlessly inefficient runtime code were improved.

Mixing Old and New Binaries

It is an overstatement to say that object files and libraries compiled by the 4.2 compiler cannot be linked with object files and libraries compiled by the 5.0 compiler. The statement is true whenever the files and libraries present a C++ interface.

Sometimes a library is coded in C++ for convenience, yet presents only a C interface to the outside world. Put simply, having a C interface means that a client cannot tell the program was written in C++. More specifically, having a C interface means that all of the following are true:

- All externally-called functions have C linkage and use only C types for parameters and returned value.
- All pointers-to-function in the interface have C linkage and use only C types for parameters and returned value.
- All externally-visible types are C types.
- All externally-available objects have C types.
- No C++ exceptions escape from or are passed into the library.
- Use of `cin`, `cout`, `cerr`, or `clog` is not permitted.

If a library meets the C-interface criteria, it can be used where ever a C library can be used. In particular, such libraries can be compiled with one version of the C++ compiler and linked with object files compiled with a different version.

However, if any of these conditions are violated, the files and libraries cannot be linked together. If an attempted link succeeds, which is doubtful, the program does not run correctly.

Conditional Expressions

The C++ standard introduced a change in the rules for conditional expressions. The difference shows up only in an expression like

e ? a : b = c

The critical issue is having an assignment following the colon when no grouping parentheses are present.

The 4.2 compiler used the original C++ rule and treats that expression as if you had written

(e ? a : b) = c

That is, the value of `c` will be assigned to either `a` or `b` depending on the value of `e`.

The 5.0 compiler in both compatibility and standard mode uses the new C++ rule. It treats that expression as if you had written

e ? a : (b = c)

That is, `c` will be assigned to `b` if and only if `e` is false.

Solution: Always use parentheses to indicate which meaning you intend. You can then be sure the code will have the same meaning when compiled by any compiler.

Function Pointers and `void*`

In C there is no implicit conversion between pointer-to-function and `void*`. The ARM added an implicit conversion between function pointers and `void*` “if the value would fit.” C++ 4.2 implements that rule. The implicit conversion was later removed from C++, since it causes unexpected function overloading behavior, and because it reduces portability of code. In addition, there is no longer any conversion, even with a cast, between pointer-to-function and `void*`.

C++ 5.0, in both compatibility mode and standard mode, now issues a warning for implicit and explicit conversions between pointer-to-function and `void*`. In both modes it no longer recognizes such implicit conversions when resolving overloaded function calls. Such code that currently compiles with the 4.2 compiler generates an error (no matching function) with the 5.0 compiler. If you have code that depends on the implicit conversion for proper overload resolution, you need to add a cast. For example:

```
int g(int);
typedef void (*fptr)();
int f(void*);
int f(fptr);
void foo()
{
    f(g);          // This line has different behavior
}
```

With the 4.2 compiler, the marked line in the code example calls `f(void*)`. With the 5.0 compiler, there is no match, and you get an error message. You can add an explicit cast, such as `f((void*)g)`, but you will get a warning because the code is invalid. You should modify your code so that you do not need to convert between function pointers and `void*`. Such code has never been portable, even when this conversion was allowed.

C++ does not have a “universal function pointer” corresponding to `void*`. With C++ on all supported platforms, all function pointers have the same size and representation. You can therefore use any convenient function pointer type to hold the value of any function pointer. This solution is portable to many, but not all, platforms. As always, you must convert the pointer value back to its original type before attempting to call the function that is pointed to. See also “Pointers to `extern "C"` Functions” on page 27.

CHAPTER 2

Using Compatibility Mode

This chapter describes using code intended for the C++ 4 compilers with the C++ 5.0 compiler.

Compatibility Mode

The compiler options for compatibility mode are (both versions mean the same thing):

```
-compat
-compat=4
```

For example:

```
CC -compat -O myfile.cc mylib.a -o myprog
```

There are some minor differences between using the C++ 4 compiler and the C++ 5.0 compiler in compatibility mode, as described in the following sections.

Keywords in Compatibility Mode

By default, some of the new C++ keywords are recognized as keywords in compatibility mode, but you can turn off most of these keywords with compiler options, as shown in the following table. Changing the source code to avoid the keywords is preferable to using the compiler options.

TABLE 2–1 Keywords in Compatibility Mode

Keyword	Compiler option to disable
explicit	-features=no%explicit
export	-features=no%export
mutable	-features=no%mutable
typename	*cannot disable*

Keyword typename cannot be disabled. The additional new C++ keywords, described in Table 3–1, are disabled by default in compatibility mode.

Language Semantics

The 5.0 compiler does a better job of enforcing some C++ language rules. It is also less permissive about anachronisms.

If you compile with C++ 4 and enable anachronism warnings, you might discover code that has always been invalid, but that much older C++ compilers accepted anyway. It was always explicit policy (that is, stated in the manuals) that the anachronisms would cease to be supported in future compiler releases. The anachronisms consist mainly of violating access (private, protected) rules, violating type-matching rules, and using compiler-generated temporary variables as the target of reference parameters.

The rules that previously were not enforced are as follows:

- When initializing an object, or passing or returning a value of class type, the copy constructor must be accessible.

```
class T {
      T(const T&);          // private
  };
  void f1(T t) { }          // Error, can’t pass a T
  T f2() { T t; return t; } // Error, can’t return a T
```

  Solution: Make the copy constructor accessible, usually public.

- The static storage class applies to objects, not to types.

```
static class C {...};   // Error, cannot use static here
  static class D {...} d; // OK, d is static
```

Solution: In this example, the `static` keyword does not have any meaning for class `C` and should be removed.

- When allocating an object with `new`, the matching operator `delete` must be accessible.

```
class T {
      void operator delete(void*); // private
    public:
      void* operator new(size_t);
  };
  T* t = new T;              // Error, operator delete is not accessible
```

Solution: Make operator delete accessible, usually public.

- Default parameter values on overloaded operators or on pointers to functions are not allowed.

```
T operator+(T t1, T t2 = T(0) );   // Error
  void (*fptr)(int = 3);             // Error
```

Solution: You must write the code some other way, probably by providing additional function or function pointer declarations.

- The function `main` must have a return type of `int`.

```
void main(){ ... } // error
```

Solution: Give `main` a return type of `int`. It is not necessary to add a return statement. If a return statement is present, it must return an `int` value.

- Trailing commas in function argument lists are not allowed.

```
f(int i, int j, ){ ... } // error
```

Solution: Remove the extra comma.

- Passing a `const` or literal value to a nonconstant reference parameter is not allowed.

```
void f(T&);
  extern const T t;
  void g() {
      f(t);  // Error
  }
```

Solution: If the function does not modify its parameter, change the declaration to take a `const` reference (for this example, `const T&`). If the function modifies the parameter, you cannot pass it a `const` or a literal value. An alternative is to create an explicit nonconstant temporary and pass that instead. See "String Literals and `char*` " on page 21 for related information.

- A count is not allowed in a delete-expression.

```
delete [5] p; // error: should be delete [] p;
```

- A version of `operator new` or `operator delete` that is not a class member must not be declared static.

```
static void* operator new(size_t); // error
```

Solution: Make the function global.

- If an object of `enum` type is initialized or assigned a value, that value must have the same `enum` type.

```
enum E { one = 1 };
  E e1 = 1;        // Error, can’t use an int to intialize an E
  E e2 = E(1);    // OK with cast
```

Solution: Use a cast.

- Macros cannot be redefined to a different value without an intervening #`undef`.

```
#define count 1
  #define count 2 // Error
```

Solution: Remove one of the `#define` statements, or put a `#undef count` statement before the second `#define`.

- The old C++ syntax of implied base-class name in a member-initializer list is not allowed.

```
struct B { B(int); };
  struct D : B {
      D(int i) : (i) { }    // Error, should be B(i)
  };
```

- If you allocate a const object with `new`, it must be initialized.

```
const int* ip1 = new const int;    // Error
  const int* ip2 = new const int(3); // OK
```

- `const` and `volatile` qualifiers on pointers must match properly when passing arguments to functions, and when initializing variables.

```
void f(char *);
  const char* p = ‘‘hello’’;
  f(p);              // Error: passing const char* to non-const char*
```

Solution: If the function does not modify the characters it points to, declare the parameter to be `const char*`. Otherwise, make a nonconstant copy of the string and pass that instead.

- Nested types cannot be accessed from outside the enclosing class without a class qualifier.

```
struct Outer {
      struct Inner { int i; };
      int j;
};
Inner x;            // Error; should be Outer::Inner
```

CHAPTER 3

Using Standard Mode

This chapter explains use of the standard mode on the C++ 5.0 compiler.

Standard Mode

Since standard mode is the primary default, no option is required. You can also choose the compiler option:

```
-compat=5
```

For example:

```
% CC -O myfile.cc mylib.a -o myprog
```

Keywords in Standard Mode

C++ has added several new keywords. If you use any of these as identifiers, you get numerous and sometimes bizarre error messages. (It is quite difficult to determine when a programmer has used a keyword as an identifier, and the compiler error messages might not be helpful in such cases.)

Most of the new keywords can be disabled with a compiler option, as shown in the following table. Some are logically related, and get enabled or disabled in a group.

TABLE 3–1 Keywords in Standard Mode

Keyword	Compiler option to disable
`bool`, `true`, `false`	`-features=no%bool`
`explicit`	`-features=no%explicit`
`export`	`-features=no%export`
`mutable`	`-features=no%mutable`
`namespace`, `using`	*cannot disable*
`typename`	*cannot disable*
`and`, `and_eq`, `bitand`, `compl`, `not`, `not_eq`, `or`, `bitor`, `xor`, `xor_eq`	`-features=no%altspell` (see note below)

Note - Alternative spellings for special tokens: The addendum to the ISO C standard introduced the C standard header `<iso646.h>`, which defined new macros to generate the special tokens. The C++ standard has introduced these spellings directly as reserved words. (When the alternative spellings are enabled, including `<iso646.h>` in your program has no net effect.) The meaning of these tokens is shown in the following table.

TABLE 3–2 Alternative Token Spellings

Token	Spelling
`&&`	`and`
`&&=`	`and_eq`
`&`	`bitand`
`~`	`compl`

TABLE 3–2 Alternative Token Spellings *(continued)*

Token	**Spelling**
`!`	`not`
`!=`	`not_eq`
`||`	`or`
`|`	`bitor`
`~`	`xor`
`~=`	`xor_eq`

Templates

The C++ standard has some new rules for templates that make old code nonconforming, particularly code involving the use of the new keyword `typename`. The 5.0 compiler does not yet enforce these rules, but does recognize this keyword. Template code that worked under the 4.2 compiler probably continues to work, although the 4.2 version accepted some invalid template code. You should migrate your code to the new C++ rules as development schedules permit, since future compilers will enforce the new rules.

Resolving Type Names

The C++ standard has new rules for determining whether an identifier is the name of a type. The following example illustrates the new rules:

```
typedef int S;
class B { ... typedef int U; ... }
template< class T > class C : public B {
    S    s; // OK
    T    t; // OK
    U    x; // 1 No longer valid
    B::U y; // 2 No longer valid
    T::V z; // 3 No longer valid
};
```

The new language rules state that no base class is searched automatically to resolve type names in a template, and that no name coming from a base class or template parameter class is a type name, unless it is declared to be so with the keyword `typename`.

The first invalid line (1) in the code example tries to inherit `U` from `B` as a type without the qualifying class name and without the keyword `typename`. The second invalid line (2) correctly modifies the type with the name of the base class, but the `typename` modifier is missing. The third invalid line (3) uses type `V` coming from the template parameter, but omits the keyword `typename`. The definition of `s` is valid because the type doesn't depend on a base class or member of a template parameter. Similarly, the definition of `t` is valid because it uses type `T` directly, a template parameter which must be a type.

The following code shows the correct implementation:

```
typedef int S;
class B { ... typedef int U; ... }
template< class T > class C : public B {
    S            s; // OK
    T            t; // OK
    typename B::U x; // OK
    typename B::U y; // OK
    typename T::V z; // OK
};
```

Converting to the New Rules

A problem for migrating code is that `typename` was not previously a keyword. If existing code uses `typename` as an identifier, you must first change the name to something else.

For code that must work with old and new compilers, you can add statements that are similar to the following example to a project-wide header file.

```
#ifdef TYPENAME_NOT_RECOGNIZED
#define typename
#endif
```

The effect is to conditionally replace `typename` with nothing. When using older compilers (such as Sun C++ 4.2) that do not recognize `typename`, add `-DTYPENAME_NOT_RECOGNIZED` to the set of compiler options in your makefile.

Explicit Instantiation and Specialization

In the ARM, and in the 4.2 compiler, there was no standard way to request an explicit instantiation of a template using the template definition. The C++ standard, and the 5.0 compiler in standard mode, provide a syntax for explicit instantiation using the template definition; the keyword `template` followed by a declaration of the type. For example, the last line in the following code forces the instantiation of class `MyClass` on type `int`, using the default template definition.

```
template<class T> class MyClass {
    ...
};
template class MyClass<int>; // explicit instantiation
```

The syntax for explicit specializations has changed. To declare an explicit specialization, or to provide the full definition, you now prefix the declaration with `template<>`. (Notice the empty angle brackets.) For example:

```
// specialization of MyClass
class MyClass<char>;         // old-style declaration
class MyClass<char> { ... }; // old-style definition
template<> class MyClass<char>;         // standard declaration
template<> class MyClass<char> { ... }; // standard definition
```

The declaration forms mean that the programmer has somewhere provided a different definition (specialization) for the template for the provided arguments, and the compiler is not to use the default template definition for those arguments.

In standard mode, the 5.0 compiler accepts the old syntax as an anachronism. The 4.2 compiler accepts the new specialization syntax, but does not treat code using the new syntax correctly in every case. (The standard changed after the feature was put into the 4.2 compiler.) For maximum portability of template specialization code, you can add statements similar to the following to a project-wide header:

```
#ifdef OLD_SPECIALIZATION_SYNTAX
#define Specialize
#else
#define Specialize template<>
#endif
```

Then you would write, for example:

Specialize class MyClass<char>; // declaration

Template Repository

The Sun implementation of C++ templates uses a repository for template instances. The C++ 4.2 compiler stored the repository in a directory called `Templates.DB`. The Sun C++ 5.0 compiler, by default, uses directories called `SunWS_cache` and `SunWS_config`. SunWS_cache contains the working files and SunWS_config contains the configuration files, specifically, the template options file (`SunWs_config/CC_tmpl_opt`). (See the *C++ Users' Guide*.)

If you have makefiles that for some reason mention repository directories by name, you need to modify the makefiles. Furthermore, the internal structure of the repository has changed, so any makefiles that access the contents of `Templates.DB` no longer work.

In addition, standard C++ programs probably make heavier use of templates. Paying attention to the considerations of multiple programs or projects that share directories is very important. If possible, use the simplest organization. You must compile only files belonging to the same program or library in any one directory. The template repository then applies to exactly one program. If you compile a different program in the same directory, clear the repository by using `CCadmin -clean`. See *C++ User's Guide* for more information.

The danger in more than one program sharing the same repository is that different definitions for the same name might be required. This situation cannot be handled correctly when the repository is shared.

Templates and the Standard Library

The C++ standard library contains many templates, and many new standard header names to access those templates. The Sun C++ standard library puts declarations in the template headers, and implementation of the templates in separate files. If one of your project file names matches the name of a new template header, it is possible to pick up the wrong implementation file, and cause numerous, bizarre errors. Suppose you have your own template called `vector`, putting the implementation in a file called `vector.cc`. Depending on file locations and command-line options, it is possible for the compiler to pick up your `vector.cc` when it needs the one from the standard library, or vice-versa. When the export keyword and exported templates are implemented in a future compiler version, the situation will be worse.

There are two recommendations for preventing current and future problems:

- Do not use any of the standard header names as names of your template files. All of the standard library is in namespace `std`, so you won't get direct name conflicts with your own templates or classes. You can still get indirect conflicts from `using` declarations or directives, so it is preferable not to duplicate template names from the standard library. The standard headers involving templates are as follows:

```
algorithm     bitset      complex      deque      exception
   fstream       functional  iomanip      ios        iosfwd
   iostream      istream     iterator     limits     list
   locale        map         memory       numeric    ostream
   queue         set         sstream      stack      stdexcept
   streambuf     string      typeinfo     utility    valarray
   vector
```

- Put template implementations in the header (`.h`) file, instead of in a separate file, to prevent implementation file name conflicts. See *C++ Users' Guide* for more information.

Class Name Injection

The C++ standard says that the name of a class is "injected" into the class itself. This is a change from earlier C++ rules. Formerly, the name of the class was not found as a name within the class.

In most cases, this subtle change has no effect on an existing program. In some cases, this change can make a formerly valid program invalid, and sometimes can result in a change of meaning. For example:

```
const int X = 5;

class X {
    int i;
public:
    X(int j = X) : // what is the default value X?
        i(j) { }
};
```

To determine the meaning of `X` as a default parameter value, the compiler looks up the name `X` in the current scope, then in successive outer scopes, until it finds an `X`:

- Under the old C++ rules, the name of the class `X` would not be found in the class scope, and the integer name `X` at file scope hides the class name `X`. The default value is therefore 5.
- Under the new C++ rules, the name of class `X` is found in the class itself. The compiler finds `X` in the class and generates an error, because the `X` it finds is a type name, not an integer value.

Because having a type and an object with the same name in the same scope is considered poor programming practice, this error should rarely occur. If you get such an error, you can fix the code by qualifying the variable with the proper scope, such as:

```
X(int j = ::X)
```

The next example (adapted from the standard library) illustrates another scoping problem.

```
template class<T> class iterator { ... };

template class<T> class list {
     public:
       class iterator { ... };
       class const_iterator : public ::iterator<T> {
         public:
            const_iterator(const iterator&); // which iterator?
};
```

What is the parameter type to the constructor for `const_iterator`? Under the old C++ rules, the compiler does not find the name `iterator` in the scope of class `const_iterator`, so it searches the next outer scope, class `list<T>`. That scope has a member type `iterator`. The parameter type is therefore `list<T>::iterator`.

Under the new C++ rules, the name of a class is inserted into its own scope. In particular, the name of a base class is inserted into the base class. When the compiler starts searching for a name in a derived class scope, it can now find the name of a base class. Since the type of the parameter to the `const_iterator` constructor does not have a scope qualifier, the name that is found is the name of the `const_iterator` base class. The parameter type is therefore the global `::iterator<T>`, instead of `list<T>::iterator`.

To get the intended result, you can change some of the names, or use a scope qualifier, such as:

const_iterator(const list<T>::iterator&);

`for`-Statement Variables

The ARM rules stated that a variable declared in the header of a `for`-statement was inserted into the scope containing the `for`-statement. The C++ committee felt that this rule was incorrect, and that the variable's scope should end at the end of the `for`-statement. (In addition, the rule didn't cover some common cases and, as a result, some code worked differently with different compilers.) The C++ committee changed the rule accordingly. Many compilers, C++ 4.2 included, continued to use the old rule.

In the following example, the `if`-statement is valid under the old rules, but invalid under the new rules, because `k` has gone out of scope.

```
for( int k = 0; k < 10; ++k ) {
    ...
}
if( k == 10 ) ...       // Is this code OK?
```

In compatibility mode, the C++ 5.0 compiler uses the old rule by default. You can instruct the compiler to use the new rule with the `-features=localfor` compiler option.

In standard mode, the C++ 5.0 compiler uses the new rule by default. You can instruct the compiler to use the old rule with the `-features=no%localfor` compiler option.

You can write code that works properly with all compilers in any mode by pulling the declaration out of the `for`-statement header, as shown in the following example.

```
int k;
for( k = 0; k < 10; ++k ) {
    ...
}
if( k == 10 ) ...      // Always OK
```

String Literals and `char*`

Some history might help you understand this subtle issue. Standard C introduced the `const` keyword and the concept of constant objects, neither of which was present in the original C language ("K&R" C). A string literal such as `"Hello world"` logically should be `const` in order to prevent nonsensical results, as in the following example.

```
#define GREETING ‘‘Hello world’’;
char* greet = GREETING; // No compiler complaint
greet[0] = ‘J’;
printf(‘‘%s’’, GREETING); // Prints ‘‘Jello world’’ on some systems
```

In both C and C++, the results of attempting to modify a string literal are undefined. The previous example produces the odd result shown if the implementation chooses to use the same writable storage for identical string literals.

Because so much then-existing code looked like the second line in the preceding example, the C Standards Committee in 1989 did not wish to make string literals `const`. The C++ language originally followed the C language rule. The C++ Standards Committee later decided that the C++ goal of type safety was more important, and changed the rule.

In standard C++, string literals are constant and have type `const char[]`. The second line of code in the previous example is not valid in standard C++. Similarly, a function parameter declared as `char*` should no longer be passed as a string literal. However, the C++ standard also provides for a deprecated conversion of a string literal from `const char[]` to `char*`. Some examples are:

```
char *p1 = ‘‘Hello’’;         // Formerly OK, now deprecated
const char* p2 = ‘‘Hello’’;  // OK
void f(char*);
f(p1);         // Always OK, since p1 is not declared const
f(p2);         // Always an error, passing const char* to char*
f(‘‘Hello’’);   // Formerly OK, now deprecated
void g(const char*);
```

```
g(p1);        // Always OK
g(p2);        // Always OK
g(‘‘Hello’’);  // Always OK
```

If a function does not modify, directly or indirectly, a character array that is passed as an argument, the parameter should be declared `const char*` (or `const char[]`). You might find that the need to add `const` modifiers propagates through the program; as you add modifiers, still more become necessary. (This phenomenon is sometimes called “const poisoning.”)

C++ 5.0 in standard mode issues a warning about the deprecated conversion of a string literal to `char*`. If you were careful to use `const` wherever it was appropriate in your existing programs, they probably compile without these warnings under the new rules.

For function overloading purposes, a string literal is always regarded as const in standard mode. For example:

```
void f(char*);
void f(const char*);
f(‘‘Hello’’); // which f gets called?
```

If the above example is compiled in compatibility mode (or with the 4.2 compiler), function `f(char*)` is called. If compiled in standard mode, function `f(const char*)` is called.

In standard mode, the compiler will put literal strings in read-only memory by default. If you then attempt to modify the string (which might happen due to automatic conversion to `char*`) the program aborts with a memory violation.

With the following example, the 4.2 compiler and the 5.0 compiler in compatibility mode put the string literal in writable memory. The program will run, although it technically has undefined behavior. The 5.0 compiler in standard mode puts the string literal in read-only memory by default, and the program aborts with a memory fault. You should therefore heed all warnings about conversion of string literals, and try to fix your program so the conversions do not occur. Such changes will ensure your program is correct for every C++ implementation.

```
void f(char* p) { p[0] = ‘J’; }

int main()
{
    f(‘‘Hello’’); // conversion from const char[] to char*
}
```

You can change the compiler behavior with the use of a compiler option:

- The `-features=conststrings` compiler option instructs the compiler to put string literals in read-only memory even in compatibility mode.
- The `-features=no%conststrings` compiler option causes the compiler to put string literals in writable memory even in standard mode.

You might find it convenient to use the standard C++ `string` class instead of C-style strings. The C++ string class does not have the problems associated with string literals, because standard `string` objects can be declared separately as `const` or not, and can be passed by reference, by pointer, or by value to functions.

New Forms of `new` and `delete`

There are four issues regarding the new forms of `new` and `delete`:

- Array forms
- Exception specifications
- Replacement functions
- Header files

The old rules are used by default in compatibility mode, and the new rules are used by default in standard mode. Changing from the default is not recommended, because the old run-time library (`libC.so`) depends on the old definitions and behavior, and the new standard library (`libCstd.so`) depends on the new definitions and behavior.

The compiler predefines the macro `_ARRAYNEW` to the value 1 when the new rules are in force. The macro is not defined when the old rules are in use. The following example is explained in more detail in the next section:

```
// Replacement functions
#ifdef _ARRAYNEW
    void* operator new(size_t) throw(std::bad_alloc);
    void* operator new[](size_t) throw(std::bad_alloc);
#else
    void* operator new(size_t);
#endif
```

Array Forms of `new` and `delete`

The C++ standard adds new forms of `operator new` and `operator delete` that are called when allocating or deallocating an array. Previously, there was only one form of these operator functions. In addition, when you allocate an array, only the global form of `operator new` and `operator delete` would be used, never a class-specific form. The C++ 4.2 compiler did not support the new forms, since their use requires an ABI change.

In addition to these functions:

void operator new(size_t);*

void operator delete(void);*

there are now:

void operator new[](size_t);*

void operator delete[](void);*

In all cases (previous and current), you can write replacements for the versions found in the run-time library. The two forms are provided so that you can use a different memory pool for arrays than for single objects, and so that a class can provide its own version of `operator new` for arrays.

Under both sets of rules, when you write `new T`, where `T` is some type, function `operator new(size_t)` gets called. However, when you write `new T[n]` under the new rules, function `operator new[](size_t)` is called.

Similarly, under both sets of rules, when you write `delete p`, `operator delete(void*)` is called. Under the new rules, when you write `delete [] p`, `operator delete[](void*)` is called.

You can write class-specific versions of the array forms of these functions, as well.

Exception Specifications

Under the old rules, all forms of `operator new` returned a null pointer if the allocation failed. Under the new rules, the ordinary forms of `operator new` throw an exception if allocation fails, and do not return any value. Special forms of `operator new` that return zero instead of throwing an exception are available. All versions of `operator new` and `operator delete` have an *exception-specification*. The declarations found in standard header `<new>` are:

```
namespace std {
      class bad_alloc;
      struct nothrow_t {};
      extern const nothrow_t nothrow;
}
// single-object forms
void* operator new(size_t size) throw(std::bad_alloc);
void* operator new(size_t size, const std::nothrow_t&) throw();
void operator delete(void* ptr) throw();
void operator delete(void* ptr, const std::nothrow_t&) throw();
// array forms
void* operator new[](size_t size) throw(std::bad_alloc);
void* operator new[](size_t size, const std::nothrow_t&) throw();
void operator delete[](void* ptr) throw();
void operator delete[](void* ptr, const std::nothrow_t&) throw();
```

Defensive code such as the following example no longer works as previously intended. If the allocation fails, the `operator new` that is called automatically from the new-expression throws an exception, and the test for zero never occurs.

```
T* p = new T;
if( p == 0 ) {           // No longer OK
    ...                  // Handle allocation failure
}
...                      // Use p
```

There are two solutions:

- Rewrite the code to catch the exception. For example:

```
T* p = 0;
   try {
       p = new T;
   }
   catch( std::bad_alloc& ) {
       ...       // Handle allocation failure
   }
   ...           // Use p
```

- Use the `nothrow` version of `operator new` instead. For example:

```
T* p = new (std::nothrow) T;
   ... remainder of code unchanged from original
```

If you prefer not to use any exceptions in your code, you can use the second form. If you are using exceptions in your code, consider using the first form.

If you did not previously verify whether `operator new` succeeded, you can leave your existing code unchanged. It then aborts immediately on allocation failure instead of progressing to some point where an invalid memory reference occurs.

Replacement Functions

If you have replacement versions of `operator new` and `delete`, they must match the signatures shown in “Exception Specifications” on page 24, including the exception specifications on the functions. In addition, they *must* implement the same semantics. The normal forms of `operator new` must throw a `bad_alloc` exception on failure; the `nothrow` version must not throw any exception, but must return zero on failure. The forms of `operator delete` must not throw any exception. Code in the standard library uses the global `operator new` and `delete` and depends on this behavior for correct operation. Third-party libraries can have similar dependencies.

The global version of `operator new[]()` in the C++ 5.0 runtime library just calls the single-object version, `operator new()`, as required by the C++ standard. If you replace the global version of `operator new()` from the C++ 5.0 standard library, you don’t need to replace the global version of `operator new[] ()`.

The C++ standard prohibits replacing the predefined "placement" forms of `operator new`:

void operator new(std::size_t, void*) throw();*

void operator new[](std::size_t, void*) throw();*

They cannot be replaced in C++ 5.0 standard mode, although the 4.2 compiler allowed it. You can, of course, write your own placement versions with different parameter lists.

Header Inclusions

In compatibility mode, include `<new.h>` as always. In standard mode, include `<new>` (no `.h`) instead. To ease in transition, a header `<new.h>` is available in standard mode that makes the names from namespace `std` available in the global namespace. This header also provides typedefs that make the old names for exceptions correspond to the new exception names. See "Standard Exceptions" on page 31.

Boolean Type

The boolean types—`bool`, `true`, and `false`—are controlled by the presence or absence of boolean keyword recognition in the compiler:

- In compatibility mode, boolean keyword recognition is off by default. You can turn on recognition of the boolean keywords with the compiler option `-features=bool`.
- In standard mode, boolean keyword recognition is on by default. You can turn off recognition of these keywords using the compiler option `-features=no%bool`.

Turning on the keywords in compatibility mode is a good idea, because it exposes any current use of the keywords in your code. (Note: Even if your old code uses a compatible definition of the boolean type, the actual type is different, affecting name mangling. You must recompile all old code using the boolean type in function parameters if you do this.)

Turning off the boolean keywords in standard mode is not a good idea, because the C++ standard library depends on the built-in `bool` type, which would not be available. When you later turn on `bool`, more problems ensue, particularly with name mangling.

The compiler predefines the macro `_BOOL` to be 1 when the boolean keywords are enabled. It is not defined when they are disabled. For example:

```
// define a reasonably compatible bool type
#if !defined(_BOOL) && !defined(BOOL_TYPE)
    #define BOOL_TYPE           // Local include guard
    typedef unsigned char bool; // Standard-mode bool uses 1 byte
    const bool true = 1;
    const bool false = 0;
#endif
```

It is not possible to define a boolean type in compatibility mode that works exactly like the new built-in `bool` type. This is one reason why a built-in boolean type was added to C++.

Pointers to `extern "C"` Functions

A function can be declared with a language linkage, such as

extern "C" int f1(int);

If you do not specify a linkage, C++ linkage is assumed. You can specify C++ linkage explicitly:

extern "C++" int f2(int);

You can also group declarations:

```
extern ‘‘C’’ {
    int g1(); // C linkage
    int g2(); // C linkage
    int g3(); // C linkage
} // no semicolon
```

This technique is used extensively in the standard headers.

Language Linkage

Language linkage means the way in which a function is called: where the arguments are placed, where the return value is to be found, and so on. Declaring a language linkage does not mean the function is written in that language. It means that the function is called *as if* it were written in that language. Thus, declaring a C++ function to have C linkage means the C++ function can be called from a function written in C.

A language linkage applied to a function declaration applies to the return type and all its parameters that have function or pointer-to-function type.

The C++ 4.2 compiler implements the ARM rule that the language linkage is not part of the function type. In particular, you can declare a pointer to a function without regard to the linkage of the pointer, or of a function assigned to it. The C++ 5.0 compiler in compatibility mode uses the same rule.

The C++ 5.0 compiler in standard mode implements the new rule that the language linkage is part of its type, and is part of the type of a pointer to function. The linkages must therefore match.

The following example shows functions and function pointers with C and C++ linkage, in all four possible combinations. The 4.2 compiler and the 5.0 compiler in compatibility mode accept all combinations. The 5.0 compiler in standard mode accepts the mismatched combinations only as an anachronism.

```
extern ‘‘C’’ int fc(int) { return 1; }      // fc has C linkage
int fcpp(int) { return 1; }                // fcpp has C++ linkage
// fp1 and fp2 have C++ linkage
int (*fp1)(int) = fc;                      // Mismatch
int (*fp2)(int) = fcpp;                    // OK
// fp3 and fp4 have C linkage
extern ‘‘C’’ int (*fp3)(int) = fc;           // OK
extern ‘‘C’’ int (*fp4)(int) = fcpp;         // Mismatch
```

If you encounter a problem, be sure that the pointers to be used with C linkage functions are declared with C linkage, and the pointers to be used with C++ linkage functions are declared without a linkage specifier, or with C++ linkage. For example:

```
extern ‘‘C’’ {
    int fc(int);
    int (*fp1)(int) = fc; // Both have C linkage
}
int fcpp(int);
int (*fp2)(int) = fcpp;   // Both have C++ linkage
```

In the worst case, where you really do have mismatched pointer and function, you can write a “wrapper” around the function to avoid any compiler complaints. (On Solaris, C and C++ function linkage is the same. However, in the most general case, the mismatched linkage really is an error. Hence the new language rule.)

In the following example, `composer` is a C function taking a pointer to a function with C linkage.

```
extern ‘‘C’’ void composer( int(*)(int) );
extern ‘‘C++’’ int foo(int);
composer( foo ); // Mismatch
```

To pass function `foo` (which has C++ linkage) to the function `composer`, create a C-linkage function `foo_wrapper` that presents a C interface to `foo`:

```
extern ‘‘C’’ void composer( int(*)(int) );
extern ‘‘C++’’ int foo(int);
extern ‘‘C’’ int foo_wrapper(int i) { return foo(i); }
composer( foo_wrapper ); // OK
```

In addition to eliminating the compiler complaint, this solution works even if C and C++ functions really have different linkage.

A Less-Portable Solution

The Sun implementation of C and C++ function linkage is binary-compatible. That is not the case with every C++ implementation, although it is reasonably common. If you are not concerned with possible incompatibility, you can employ a cast to use a C++-linkage function as if it were a C-linkage function.

A good example concerns static member functions. Prior to the new C++ language rule regarding linkage being part of a function’s type, the usual advice was to treat a static member function of a class as a function with C linkage. Such a practice circumvented the limitation that you cannot declare any linkage for a class member function. You might have code like the following:

```
// Existing code
typedef int (*cfuncptr)(int);
extern ‘‘C’’ void set_callback(cfuncptr);
class T {
    ...
    static int memfunc(int);
};
...
set_callback(T::memfunc); // no longer valid
```

As recommended in the previous section, you can create a function wrapper that calls `T::memfunc` and then change all the `set_callback` calls to use a wrapper instead of `T::memfunc`. Such code will be correct and completely portable.

An alternative is to create an overloaded version of `set_callback` that takes a function with C++ linkage and calls the original, as in the following example:

```
// Modified code
extern ‘‘C’’ {
    typedef int (*cfuncptr)(int); // ptr to C function
    void set_callback(cfuncptr);
}
typedef int (*cppfuncptr)(int); // ptr to C++ function
inline void set_callback(cppfuncptr f) // overloaded version
    { set_callback((cfuncptr)f); }
class T {
    ...
    static int memfunc(int);
};
...
set_callback(T::memfunc); // unchanged from original code
```

This example requires only a small modification to existing code. An extra version of the function that sets the callback was added. Existing code that called the original `set_callback` now calls the overloaded version that in turn calls the original version. Since the overloaded version is an inline function, there is no runtime overhead at all.

Although this technique works with Sun C++, it is not guaranteed to work with every C++ implementation, since the calling sequence for C and C++ functions may be different on other systems.

Pointers to Functions as Function Parameters

A subtle consequence of the new rule for language linkage involves functions that take pointers to functions as parameters, such as

```
extern ‘‘C’’ void composer( int(*)(int) );
```

:.

An unchanged rule about language linkage is that if you declare a function with language linkage, and follow it with a definition of the *same function* with no language linkage specified, the previous language linkage applies. For example:

```
extern ‘‘C’’ int f(int);
int f(int i) { ... } // Has ‘‘C’’ linkage
```

In this example, function `f` has C linkage. The definition that follows the declaration (the declaration might be in a header file that gets included) inherits the linkage specification of the declaration. But suppose the function takes a parameter of type pointer-to-function, as in the following example:

```
extern ‘‘C’’ int g( int(*)(int) );
int g( int(*pf)(int) ) { ... } // Is this ‘‘C’’ or ‘‘C++’’ linkage?
```

Under the old rule, and with the 4.2 compiler, there is only one function `g`. Under the new rule, the first line declares a function `g` with C linkage that takes a pointer-to-function-with-C-linkage. The second line defines a function that takes a pointer-to-function-with-C++-linkage. The two functions are not the same; the second function has C++ linkage. Because linkage is part of the type of a pointer-to-function, the two lines refer to a pair of overloaded functions each called `g`. Code that depended on these being the same function breaks. Very likely, the code fails during compilation or linking.

It is a good programming practice to put the linkage specification on the function definition as well as on the declaration:

```
extern ‘‘C’’ int g( int(*)(int) );
extern ‘‘C’’ int g( int(*pf)(int) ) { ... }
```

You can further reduce confusion about types by using a typedef for the function parameter:

```
extern ‘‘C’’ {typedef int (*pfc)(int);} // ptr to C-linkage function
extern ‘‘C’’ int g(pfc);
extern ‘‘C’’ int g(pfc pf) { ... }
```

Runtime Type Identification (RTTI)

In compatibility mode, RTTI is off by default, as with the 4.2 compiler. In standard mode, RTTI is on and cannot be turned off. Under the old ABI, RTTI has a noticeable cost in extra data size and in compilation efficiency. (RTTI could not be implemented directly under the old ABI, and an inefficient indirect method was required.) In standard mode, using the new ABI, RTTI has negligible cost. (This is one of several improvements in the ABI.)

Standard Exceptions

The C++ 4.2 compiler used the names related to standard exceptions that appeared in the C++ draft standard at the time the compiler was prepared. The names in the C++ standard have changed since then. The C++ 5.0 compiler in standard mode uses the standard names, as shown in the following table.

TABLE 3–3 Exception-Related Type Names

Old name	Standard name	Description
`xmsg`	`exception`	Base class for standard exceptions
`xalloc`	`bad_alloc`	Thrown by failed allocation request
`terminate_function`	`terminate_handler`	Type of a terminate handler function
`unexpected_function`	`unexpected_handler`	Type of an unexpected-exception handler function

The public members of the classes (`xmsg` vs. `exception`, and `xalloc` vs. `bad_alloc`) are different, as is the way you use the classes.

CHAPTER **4**

Using Iostreams and Library Headers

This chapter explains the library and header file changes in C++ 5.0.

Iostreams

The C++ 4.2 compiler implements *classic* iostreams, which never had a formal definition. The implementation is compatible with the version released with `Cfront` (1990), with some bug fixes.

Standard C++ defines a new and expanded iostreams (*standard* iostreams). It is better defined, feature-rich, and supports writing internationalized code.

With the C++ 5.0 compiler in compatibility mode, you get classic iostreams, the same version supplied with the C++ 4.2 compiler. Any existing iostream code that works with the 4.2 compiler should work exactly the same way with the 5.0 compiler in compatibility mode.

In standard mode, you get standard iostreams by default. If you use the standard form of header names (without "`.h`"), you get the standard headers, with all declarations in namespace `std`.

Each of the standard headers is also provided in a form ending with "`.h`" that makes the header names available in the global namespace via using-declarations. These headers are a Sun extension, and code that depends on them might not be portable. These headers allow you to compile existing (simple) iostream code without having to change the code, even though standard iostreams are used instead of classic iostreams. For example, the following example compiles using standard `iostream` name streams.

```
#include <iostream> int main() { std::cout << "Hello, world!" << std::endl; }
```

The following example, compiles with using classic `iostream` name forms.

```
#include <iostream.h> int main() { cout << "Hello, world!" << endl; }
```

Not all classic iostream code is compatible with standard iostreams. If your classic iostream code does not compile, you must either modify your code, or use classic iostreams entirely.

To use classic iostreams in standard mode, use the compiler option `library=iostream` on the CC command line. When this option is used, a special directory is searched that contains the classic iostream header files, and the classic iostream runtime library is linked with your program. You must use this option on all compilations that make up your program as well as on the final link phase, or you will get inconsistent program results.

Note - Mixing old and new forms of iostreams—including the standard input and output streams `cin`, `cout`, and `cerr`—in the same program can cause severe problems and is not recommended.

Task (Coroutine) Library

The coroutine library, accessed via the `<task.h>` header, is no longer supported. Compared to the coroutine library, Solaris threads are better integrated into the language development tools (particularly the debugger) and the operating system.

Rogue Wave Tools.h++

The Sun C++ 4.2 compiler was delivered with Rogue Wave Tools.h++ Version 7 as the default, and with Tools.h++ Version 6 for compatibility with earlier compiler releases.

The C++ 5.0 compiler does not provide Tools.h++ Version 6; it supplies Tools.h++ Version 7. Tools.h++ Version 7 is the default for the C++ 5.0 compiler in both compatibility mode and standard mode.

The Rogue Wave `Tools.h`++ version 7 library is built with classic iostreams. Therefore, when you include the Rogue Wave tools library in standard mode, you must also include `libiostream`. However, you must be careful to not use the old and new forms of iostreams—including the standard input and output streams `cin`, `cout`, and `cerr`—in the same program. Doing so can cause severe problems and is not recommended.

To use the Rogue Wave Tools.h libraries in standard mode, use the following compiler option:

- `-library=rwtools7,iostream`

To use the Rogue Wave Tools.h libraries in compatibility mode, use the following compiler option:

- `-library=rwtools7`

Refer to *C++ User's Guide* or the `CC.1` man page for more information about accessing Tools.h++.

C Library Headers

In compatibility mode, you use the standard headers from C in exactly the same way as before. The headers are in the `/usr/include` directory, supplied with the version of Solaris you are using.

For clarification, the headers being discussed are the 17 headers defined by the ISO C standard (ISO 9899:1990) plus its later addendum (1994):

```
<assert.h> <ctype.h>  <errno.h>  <float.h>  <iso646.h> <limits.h>
<locale.h> <math.h>   <setjmp.h> <signal.h> <stdarg.h> <stdio.h>
<stdlib.h> <string.h> <time.h>   <wchar.h>  <wctype.h>
```

The hundreds of other headers that reside in and below the `/usr/include` directory are not affected by this language change because they are not part of the C language standard.

The C++ standard has changed the definition of the standard C headers.

You can include and use any of these headers in a C++ program the same as in previous versions of Sun C++, but some restrictions apply.

The C++ standard requires that the names of types, objects, and functions in these headers appear in namespace `std` as well as in the global namespace. This in turn means that the versions of these headers supplied with Solaris (including Solaris 7) cannot be used directly. If you compile in standard mode, you must use the versions of these headers that are supplied with the C++ 5.0 compiler. If you use the wrong headers, your program can fail to compile or link.

Just one restriction applies: Use the standard spelling for the header, not a path name. For example, write:

```
#include <stdio.h> // Correct
```

and not either of these:

```
#include "/usr/include/stdio.h" // Wrong #include </usr/include/stdio.h> // Wrong
```

The C++ standard also introduces a second version of each of the 17 standard C headers. For each header of the form `<NAME.h>`, there is an additional header of the form `<cNAME>`. That is, the trailing "`.h`" is dropped, and a leading "`c`" is added. Some examples: `<cstdio>`, `<cstring>`, `<cctype>`.

These headers contain the names from the original form of the header, but appear only in namespace `std`. An example of use according to the C++ standard is:

```
#include <cstdio>
int main() {
    printf(‘‘Hello, ‘‘);       // Error, printf unknown
    std::printf(‘‘world!\n’’); // OK
}
```

Because the code uses `<cstdio>` instead of `<stdio.h>`, the name `printf` appears only in namespace `std` and not in the global namespace. You must either qualify the name `printf`, or add a *using-declaration*:

```
#include <cstdio>
using std::printf;
int main() {
    printf(‘‘Hello, ‘‘);       // OK
    std::printf(‘‘world!\n’’); // OK
}
```

The standard C headers in `/usr/include` contain many declarations that are not allowed by the C standard. The declarations are there for historical reasons, primarily because Unix systems have traditionally had the extra declarations in those headers, or because other standards (like POSIX or XOPEN) require them. For continued compatibility, these extra names continue to appear in the Sun C++ versions of the `<NAME.h>` headers, but only in the global namespace. These extra names do not appear in the `<cNAME>` versions of the headers. Since these new headers have never been used in any previous program, there is no compatibility or historical issue. Consequently, you might not find the `<cNAME>` headers to be useful for general programming. If you want to write maximally portable standard C++ code, however, be assured that the `<cNAME>` headers do not contain any unportable declarations. The following example uses `<stdio.h>`:

```
#include <stdio.h>                     // stdin in global namespace
int f() { return fileno(stdin); }      // OK
int g() { return std::fileno(stdin); } // Error
```

```
#include <cstdio>                            // stdin in namespace std
int f() { return fileno(std::stdin); }      // Error
int g() { return std::fileno(std::stdin); } // Error
```

The following example uses `<cstdio>`:

Function `fileno` is an extra function that for compatibility continues to appear in `<stdio.h>`, but only in the global namespace, not in namespace `std`. Because it is an extra function, it does not appear in `<cstdio>` at all.

Note - The C++ 5.0 version of the C standard headers is not fully compliant in all these respects on Solaris 2.5.1 or 2.6.

The C++ standard allows using both the `<NAME.h>` and `<cNAME>` versions of the standard C headers in the same compilation unit. Although you probably would not do this on purpose, it can happen when you include, for example, `<cstdlib>` in your own code, and some project header you use includes `<stdlib.h>`. On Solaris environments 2.5.1 and 2.6, this mixing does not work for some headers, particularly for the `<wchar.h>`/`<cwchar>` and `<wctype.h>`/`<cwctype>` header pairs. If you get compiler complaints about one of these headers, use the `<NAME.h>` version of the header in your code instead of the `<cNAME>` version.

Standard Header Implementation

The *C++ User's Guide* explains in detail how standard headers are implemented along with the reasons for the implementation method. When you include any of the standard C or C++ headers, the compiler actually searches for a file with the specified name suffixed by "`.SUNWCCh`". For example, `<string>` causes a search for `<string.SUNWCCh>` and `<string.h>` causes a search for `<string.h.SUNWCCh>`. The compiler's `include` directory contains both spellings of the names, and each pair of spellings refers to the same file. For example, in directory `include/CC` you find both `string` and `string.SUNWCCh`. They refer to the same file, the one you get when you include `<string>`.

In error messages and debugger information, the suffix is suppressed. If you include `<string>`, error message and debugger references to that file mention `string`. File dependency information uses the name `string.SUNWCCh` to avoid problems with default makefile rules regarding unsuffixed names. If you want to search for just header files (using the SunOS `find` command, for example) you can look for the `.SUNWCCh` suffix.

CHAPTER **5**

Migrating From C++ 3.0 to C++ 5.0

This chapter discusses the migration of working code from a C++ 3.0 or 3.0.1 compiler directly to the C++ 5.0 compiler.

Keywords Added Since the C++ 3.0 Compiler

The following keywords have been added to C++ since the C++ 3.0 compiler. If you use any of these as identifiers, you should change the names. As shown in Table 3–1, some keywords can be turned off in C++ 5.0.

```
bool, false, true
const_cast, dynamic_cast, reinterpret_cast, static_cast
explicit
export
mutable
namespace, using
typename
wchar_t
```

Source Code Incompatibilities

The following list describes changes that must be made to code that was written for the C++ 3.0 compiler before compiling the code with the C++ 5.0 compiler.

- K&R-style function definitions are no longer allowed. You must use prototype-style definitions.

```
int f(a) int a; { ... } // Error in C++ 5.0
```

- You cannot set a global variable `_new_handler` by assignment. Call function `set_new_handler()` instead.

- Global `operator new()` is always used when there is no in-class version. C++ 3.0 incorrectly used an outer-class version in preference to the global version. In the following example, C++ 3.0 would incorrectly use `Outer::operator new` to allocate space.

```
class Outer {
  public:
      void* operator new(size_t);
      class Inner {
      ...         // No operator new
      };
  };
  Outer::Inner* p = new Outer::Inner; // Which operator new?
```

- Typedef names cannot be used as struct (or class or union) tags. For example:

```
typedef struct { int x; } S;
  struct S b;       // OK in C++ 3.0, error in C++ 5.0
  S c;              // Always OK
```

Use tags on structures (and classes and unions). The simplest way to fix the earlier example is to use the `typedef` name also as the tag. Such code is allowed in both C and C++:

```
typedef struct S { int x; } S;
  struct S b;  // Always OK
  S c;         // Always OK
```

- You cannot redefine a name from an outer scope once it has been used in a class. Such redefinition is disallowed by the C++ standard, since it can be disastrous, but the C++ 3.0 compiler did not detect the situation. The C++ 5.0 compiler makes it an error. For example:

```
typedef int T;
  class C {
      T iv;             // type int
      typedef float T;  // redefine T -- error in C++ 5.0
      T fv;             // type float
  };
```

The solution is to change the name of one of the definitions of T.

- The C++ 3.0 compiler had a bug that allowed a pointer to a function taking unspecified parameters to act in some circumstances as a “universal” pointer-to-function, as in the following example. The C++ rule is that function pointer types must match.

```
typedef (*pfp)(int,char);
   typedef (*ufp)(...);
   int foo(int,char);
   pfp p = (ufp)foo; // Allowed by 3.0, error in 5.0
```

- Comma-expressions are not allowed in null pointer constants. Although a literal zero is a null pointer constant, an expression such as `(`*anything*`, 0)` is not:

```
int f();
   char* g()
   {
         return (f(), 0); // OK in 3.0, error in 5.0
        // should be:
        //     return (f(), (char*)0); // OK
        // or two statements:
        //    f();
        //    return 0;
   }
```

- Classes with base classes cannot be initialized with aggregate-initialization syntax. The C++ 3.0 compiler allowed this if no virtual functions were present. You should write a constructor for the class instead.

```
struct Base { int i; };
   struct Derived : Base { int j; };
   Derived d = {1, 2}; // OK with 3.0, error in 5.0
```

`Cfront` Link-Time Instantiation

The Sun C++ compiler's implementation of templates is different from that of AT&T's `Cfront` compiler. `Cfront` uses link-time instantiation, which uses the following algorithm:

1. Compile all user source files.
2. Using the prelinker, `ptlink`, link all object files created in Step 1 into a partially linked executable.
3. Examine the link output and instantiate all undefined functions for which there are matching templates.
4. Link all created templates along with the partially linked executable files from Step 2.
5. As long as there are undefined functions for which there are matching template functions, repeat Steps 3 through 4.
6. Perform the final pass of the link phase on all created object files.

The main advantage of link-time instantiation is that no special outside support is required to handle specializations (user-provided functions intended to override instantiated template functions). Only those functions that have not been defined in the user source files become targets of instantiation by the compiler.

The two main disadvantages of link-time instantiation are:

- Because an instantiation takes place during the link phase, all error messages resulting from an instantiation are deferred until *after* its use. As a result, there is no helpful traceback of where the error might have occurred.
- Repetitive calls to the prelinker can dramatically increase the link time.

CHAPTER **6**

Moving From C to C++

This chapter describes how to move programs from C to C++.

C programs generally require little modification to compile as C++ programs. C and C++ are link compatible; you don't have to modify compiled C code to link it with C++ code. See *The C++ Programming Language*, by Margaret A. Ellis and Bjarne Stroustrup, for more specific information on the C++ language.

Reserved and Predefined Words

Table 6–1 shows all reserved keywords in C++ and C, plus keywords that are predefined by C++. Keywords that are reserved in C++ but not in C are shown in **boldface.**

TABLE 6–1 Reserved Keywords

asm	do	if	return	typedef
auto	double	**inline**	short	**typeid**
bool	**dynamic_cast**	int	signed	typename
break	else	long	sizeof	union
case	enum	**mutable**	static	unsigned

TABLE 6–1 Reserved Keywords *(continued)*

catch	**explicit**	**namespace**	**static_cast**	**using**
char	**export**	**new**	struct	**virtual**
class	extern	**operator**	switch	void
const	**false**	**private**	**template**	volatile
const_cast	float	**protected**	**this**	**wchar_t**
continue	for	**public**	**throw**	while
default	**friend**	register	**true**	
delete	goto	**reinterpret_cast**	**try**	

_ _STDC_ _ is predefined to the value 0. For example:

```
#include <stdio.h>
main()
{
    #ifdef __STDC__
        printf("yes\n");
    #else
        printf("no\n");
    #endif

    #if    __STDC__ ==0
        printf("yes\n");
    #else
        printf("no\n");
    #endif
}
```

produces:

```
yes
yes
```

The following table lists reserved words for alternate representations of certain operators and punctuators specified in the current ANSI/ISO working paper from the ISO C++ Standards Committee. These alternate representations have not yet been implemented in the C++ compiler, but in future releases may be adopted as reserved words and should not be otherwise used.

TABLE 6–2 Reserved Words for Operators and Punctuators

and	**bitor**	**not**	**or**	**xor**
and_eq	**compl**	**not_eq**	**or_eq**	**xor_eq**
bitand				

Creating Generic Header Files

K&R C, ANSI C, and C++ require different header files. To make C++ header files conform to K&R C and ANSI C standards so that they are generic, use the macro `_ _cplusplus` to separate C++ code from C code. The macro `_ _STDC_ _` is defined in both ANSI C and C++. Use this macro to separate C++ or ANSI C code from K&R C code.

You can insert an `#ifdef` statement in your code to conditionally compile C++ or C using the C++ compiler. To do this, use the `_ _cplusplus` macro:

```
#ifdef __cplusplus
extern "C" int myfunc(int); // C++ declaration
#else
int myfunc();               /* K&R C declaration */
#endif
```

Note - In the past, this macro was `c_plusplus`, which is no longer accepted.

Linking to C Functions

The compiler encodes C++ function names to allow overloading. To call a C function, or a C++ function “masquerading” as a C function, you must prevent this encoding. Do so by using the `extern "C"` declaration. For example:

```
extern "C" {
double sqrt(double); //sqrt(double) has C linkage
    }
```

This linkage specification does not affect the semantics of the program using `sqrt(),` but simply causes the compiler to use the C naming conventions for `sqrt()`.

Only one of a set of overloaded C++ functions can have C linkage. You can use C linkage for C++ functions that you intend to call from a C program, but you would only be able to use one instance of that function.

You cannot specify C linkage inside a function definition. Such declarations can only be done at the global scope.

Index