C++ Migration Guide

Sun™ ONE Studio 8

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 817-0925-10
May 2003, Revision A

Send comments about this document to: docfeedback@sun.com

Contents

Tables

Code Samples

Before You Begin

This book explains what you need to know to move from 4.0, 4.0.1, 4.1, or 4.2 versions of the C++ compiler. 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. This manual is intended for programmers with a working knowledge of C++ and some understanding of the Solaris™ operating environment and UNIX® commands.

Typographic Conventions

TABLE P-1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type **`rm`** *filename.*

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*`n`*`]`	`O4, O`
{ }	Braces contain a set of choices for a required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*`dir`*`[:`*`dir`*`]`	`R/local/libs:/U/a`
...	The ellipsis indicates omission in a series.	`xinline=`*`f1`*`[,...`*`fn`*`]`	`xinline=alpha,dos`

Shell Prompts

Shell	Prompt
C shell	*machine-name*%
C shell superuser	*machine-name*#
Bourne shell and Korn shell	$
Superuser for Bourne shell and Korn shell	#

Accessing Compiler Collection Tools and Man Pages

The compiler collection components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the compilers and tools, you must have the compiler collection component directory in your `PATH` environment variable. To access the man pages, you must have the compiler collection man page directory in your `MANPATH` environment variable.

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` variable and `MANPATH` variables to access this release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Sun ONE Studio compiler collection components are installed in the `/opt` directory. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Accessing the Compilers and Tools

Use the steps below to determine whether you need to change your `PATH` variable to access the compilers and tools.

▼ To Determine Whether You Need to Set Your `PATH` Environment Variable

1. **Display the current value of the `PATH` variable by typing the following at a command prompt.**

```
% echo $PATH
```

2. **Review the output to find a string of paths that contain `/opt/SUNWspro/bin/`.**

   If you find the path, your `PATH` variable is already set to access the compilers and tools. If you do not find the path, set your `PATH` environment variable by following the instructions in the next procedure.

▼ To Set Your PATH Environment Variable to Enable Access to the Compilers and Tools

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `PATH` environment variable.**

   `/opt/SUNWspro/bin`

Accessing the Man Pages

Use the following steps to determine whether you need to change your `MANPATH` variable to access the man pages.

▼ To Determine Whether You Need to Set Your MANPATH Environment Variable

1. **Request the `dbx` man page by typing the following at a command prompt.**

   ```
   % man dbx
   ```

2. **Review the output, if any.**

   If the dbx(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next procedure for setting your `MANPATH` environment variable.

▼ To Set Your MANPATH Environment Variable to Enable Access to the Man Pages

1. **If you are using the C shell, edit your home `.cshrc` file. If you are using the Bourne shell or Korn shell, edit your home `.profile` file.**
2. **Add the following to your `MANPATH` environment variable.**

   `/opt/SUNWspro/man`

Accessing Compiler Collection Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html`.

  If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`[sm] web site. The following titles are available through your installed software only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
- The release notes are available from the `docs.sun.com` web site.

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document and does not endorse and is not responsible or liable for any content, advertising, products, or other materials on or available from such sites or resources. Sun will not be responsible or liable for any damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Release notes	HTML at `http://docs.sun.com`

Related Compiler Collection Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html` and `http://docs.sun.com`. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Document Title	Description
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Commercially Available Books

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

The C++ Programming Language 3rd edition, Bjarne Stroustrup (Addison-Wesley, 1997).

The C++ Standard Library, Nicolai Josuttis (Addison-Wesley, 1999).

Generic Programming and the STL, Matthew Austern (Addison-Wesley, 1999).

Standard C++ IOStreams and Locales, Angelika Langer and Klaus Kreft (Addison-Wesley, 2000).

Thinking in C++, Volume 1, Second Edition, Bruce Eckel (Prentice Hall, 2000).

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).

Efficient C++: Performance Programming Techniques, Dov Bulka and David Mayhew (Addison-Wesley, 2000).

Resources for Developers

Visit `http://www.sun.com/developers/studio` and click the Compiler Collection link to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of compiler collection components, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

You can find additional resources for developers at `http://www.sun.com/developers/`.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

`http://www.sun.com/service/contacting`

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Email your comments to Sun at this address:

`docfeedback@sun.com`

Please include the part number (817-0925-10) of the document in the subject line of your email.

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," and the C++ 5.0, 5.1, 5.2, 5.3, 5.4, and 5.5 compilers are referred to collectively as "C++ 5." To a large degree, C++ source code that compiled and ran under C++ 4 continues to work under the C++ 5 compilers, with a few exceptions that are due to changes in the C++ language definition. The compiler provides a compatibility mode (`-compat[=4]`) that allows nearly all of your C++ 4 code to continue to work unchanged.

Note – Object code that is compiled in standard mode (the default mode) using version 5.0, version 5.1, version 5.2, version 5.3, version 5.4, or version 5.5 of the C++ compiler is not compatible with C++ code from any earlier compiler. You can sometimes use self-contained libraries of older object code with the 5.0, 5.1, 5.2, 5.3, 5.4, and 5.5 compiler versions. The details are covered in Section 1.3, "Binary Compatibility Issues" on page 1-4.

1.1 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 or binary incompatibility.

C++ is now the subject of an international standard, ISO/IEC 14882:1998 *Programming Languages — C++*. The C++ 5.4 compiler in standard mode implements nearly all of the language as specified in the standard. The readme file that accompanies the current release describes 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++ compiler, as an extension, provides a header `<iostream.h>` which 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 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 Section 1.3, “Binary Compatibility Issues” on page 1-4.

1.2 Compiler Modes of Operation

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

1.2.1 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, in standard mode, the C++ 5 compilers use 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 Section 1.3, “Binary Compatibility Issues” on page 1-4.

You should update your code to compile in 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.

1.2.2 Compatibility Mode

To provide a migration path from C++ 4 to standard mode, the compiler provides a compatibility mode (`-compat`[=4]). The compatibility mode is fully binary-compatible and mostly source-compatible with the C++ 4 compilers. (*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 Solaris 7 and Solaris 8 operating environments on IA and SPARC platforms, but not for SPARC V9 (64-bit) processors.

Note – In this document, the term "IA" refers to the Intel 32-bit processor architecture, which includes the Pentium, Pentium Pro, and Pentium II, Pentium II Xeon, Celeron, Pentium III, and Pentium III Xeon processors and compatible microprocessor chips made by AMD and Cyrix.

Reasons to use compatibility mode:

- You have C++ object libraries compiled with a C++ 4 compiler and you can't recompile them in standard mode. (For example, you don't have the source code.)
- You need to ship a product immediately, and your source code won't compile in standard mode.

Note – Under most conditions, you cannot link object files and libraries compiled in compatibility mode (`-compat`[=4]) with object files and libraries compiled in standard mode (the default mode). For more information, see Section 1.4, "Mixing Old and New Binaries" on page 1-5.

1.3 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 the Solaris operating environment is the same as the basic Solaris ABI, which is the ABI for the C language.

1.3.1 Language Changes

C++ introduced many features (such as class member functions, overloaded functions and operators, type-safe linkage, exceptions, and templates) that 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. When the C++ standard was published, Sun designed a new ABI that allows the full C++ language to be implemented. The C++ 5 compilers use 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,” which 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.

1.4 Mixing Old and New Binaries

It is an overstatement to say that you cannot link old binaries (object files and libraries compiled by the C++ 4 compiler or compiled by the C++ 5 compilers in compatibility mode) with new binaries (object files and libraries compiled by the C++ 5 compilers in standard mode). It is possible to do this on SPARC platforms by using the `libExbridge` library.

Note – Mixing modes using `libExbridge` is supported only on SPARC platforms, not on x86 platforms. Allowing `libExbridge` to work on x86 platforms would require an ABI change and recompiling code. If you can recompile the code, you do not need `libExbridge`. We very strongly recommend that you do not mix compat modes in one program. Compile all code in standard mode instead. It is much better in both the short term and in the long term. The problem you are trying to solve by mixing modes will not go away, and will not be reliably solved in the long term by using `libExbridge`.

1.4.1 Getting Started

Install the latest `SUNWlibC` patch on all systems where the application is built or run. The `libExbridge.so.1` library in the new patch depends on the versions of `libC.so.5` and `libCrun.so.1` in the same patch. Therefore, it will not work if you just copy `libExbridge.so.1` onto a system instead of installing the patch.

1.4.2 Requirements

You can link old binaries with new binaries (as defined above) under the following conditions:

1.4.2.1 Using Exceptions

When the code uses exceptions, meaning that the code contains the `throw` or `catch` keywords (including an exception specification on a function), the requirements are as follows.

- If both standard-mode code and compatibility-mode code use exceptions, the code works as long as all active functions from the throw point up to and including the catch point are compiled in the same mode. In other words, when walking up the stack from the throw point to the catch-clause, all the C++ functions that have catch blocks must be compiled in the same mode.
- You must not link the libraries `libC` and `libCrun` statically. You must link the shared (`.so`) version of the libraries, which is the compiler default. Linking `libExbridge.so.1`, automatically links `libCrun.so.1` and `libC.so.5`. See "Using the `libExbridge` Library" on page 1-6.
- You should not create a shared library with the `-Bsymbolic` linker option or link a library that is built with the `-Bsymbolic` linker option.

  If the version of the C++ compiler that you are using supports linker scoping with `-xldscope`, use that feature to control the visibility of symbols in the library instead. See `-xldscope` in the *C++ User's Guide* or CC(1) for details.

  If the version of the C++ compiler that you are using does not support linker scoping, use linker mapfiles to control the visibility of symbols in the library. See the *Linker and Libraries Guide* for details.

1.4.2.2 Using the `libExbridge` Library

The preferred method for using `libExbridge` is to link it in with the code. To do this, add the `-lExbridge` option to your `CC` command. The `-l` option prepends `lib` to the library name.

If you cannot link with `libExbridge`, follow these instructions:

- Preload the library as follow for the C shell:

```
example% setenv LD_PRELOAD /usr/lib/libExbrige.so.1
example% my_application arg1 arg2
```

- Preload the library as follows for the Bourne or Korn shell. Note that there is no semicolon after setting `LD_PRELOAD`:

```
$ LD_PRELOAD=/usr/lib/libExbrige.so.1 my_application arg1 arg2
```

Once `LD_PRELOAD` is set, it affects all programs started afterward. Preloading `libExbridge` can cause a subsequent invocation of the shell, or a program that spawns a shell, to fail. `/usr/bin/sh` defines its own `malloc` which is not properly initialized at the time it is called by the `.init` section of the preloaded library.

Thus, when using the C shell, it is best to set the environment variable in a shell script that runs the application. The Bourne and Korn shell syntax shown creates the environment variable only for the command on the same line.

If you preload `libExbridge` and your application spawns a shell, it is possible for the application to fail. In that event, you must relink the application using `libExbridge` instead of preloading the library.

1.4.3 Structuring the Interface

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 values.
- 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.
- Use of `cin`, `cout`, `cerr`, or `clog` is not permitted.

If a library meets the C-interface criteria, it can be used wherever 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, provided they do not mix exception handling.

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.

Note that if you use the C compiler (`cc`) to link an application with a C-interface library, and if that library needs C++ run-time support, then you must create a dependency on either `libC` (compatibility mode) or `libCrun` (standard mode) using one of the following methods. If the C-interface library does not need C++ run-time support, then you do not need to link with `libC` or `libCrun`.

- **Archived C-Interface Library.** When providing an archived C-interface library, you must provide instructions on how to use the library.
  - **Standard Mode.** If the C-interface library was built with the C++ compiler (`CC`) in standard mode (the default), then the user must add `-lCrun` to the `cc` command line when using the C-interface library.

    - **Compatibility Mode.** If the C-interface library was built with the C++ compiler (`CC`) in compatibility mode (`-compat`), then the user must add `-lC` to the `cc` command line when using the C-interface library

- **Shared C-Interface Library.** When providing a shared C-interface library, you must create a dependency on `libC` or `libCrun` at the time that you build the library. When the shared library has the correct dependency, you do not need to add `-lC` or `-lCrun` to the `cc` command line when you use the library.
    - **Standard Mode.** If the C-interface library is being built in standard mode (the default mode), add `-lCrun` to the `CC` command line when you build the library.
    - **Compatibility Mode.** If the C-interface library is being built in compatibility mode (`-compat`), add `-lC` to the `CC` command line when you build the library.

1.5 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 compiler now uses the new C++ rule in both compatibility and standard mode. 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.

1.6 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 implemented 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*`.

The compiler now issues a warning for implicit and explicit conversions between pointer-to-function and `void*`. In standard mode, the compiler no longer recognizes such implicit conversions when resolving overloaded function calls. Such code that compiled with the 4.2 compiler now generates an error (no matching function) in standard mode. (The compiler emits an anachronism warning in compatibility mode.) 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*)`. Now, in standard mode, 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 violates the C++ standard. Conversions between function pointers and `void*` are valid on all versions of the Solaris operating environment, but are not portable to all platforms.

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 most 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 Section 3.11, “Pointers to `extern "C"` Functions” on page 3-18.

1.7 Anticipating Future Mangling Changes

There are some instances where the compiler does not meet the C++ standard regarding declarations that refer to the same entry. In these instances, your program will not get the correct linking behavior. To avoid this problem, follow these rules. When the mangling problem is fixed in a later release, the names will still be mangled in the same way.

- Don't use gratuitous `const` keywords in function declarations.

  Declaring a value parameter `const` is not supposed to have any effect on the function signature or on how the function can be called, so don't declare it `const`.

```
int f(const int); // the const has no meaning, don't use it
int f(int);       // do this instead
int f(const int i) { ... } // don't do this
int f(int i) { ... }       // do this instead
```

- Don't use both a `typedef` and its expanded form in any one function declaration.

```
typedef int int32;
int* foo(int*, int32*); // don't do this
// don't use both int* and int32* in the same function declaration
// write one of the following consistently instead
int* foo(int*, int*);
int32* foo(int32*, int32*);
```

- Use only `typedefs` for parameters or return types that are pointer-to-function.

```
void function( void (*)(), void (*)() ); // don't do this
typedef void (*pvf)();
void function( pvf, pvf ); // do this instead
```

- Don't use `const` arrays in function declarations.

```
void function( const int (*)[4] ); // don't use this
```

  Unfortunately, there is no direct workaround for this declaration.

If you can't avoid code that is affected by this mangling problem, for example because it occurs in headers or libraries that you don't own, you can use weak symbols to equate a declaration with its definition, as shown in the following example.

```
int cpp_function( int arg ) { return arg; }
#pragma_weak "__1c_missing_mangled_name" = cpp_function
```

You must use the mangled name versions in these types of declarations.

1.7.1 Symptoms of Improper Mangling

The compiler does not always mangle names consistently when your code has any of the features described in Section 1.7, "Anticipating Future Mangling Changes" on page 1-10 as problem areas. The symptom is that the program fails to link and the linker complains that a symbol cannot be found. The unmangled name in the linker error message refers to a function or object that is, in fact, defined. However, because the compiler mangled a reference to the symbol differently from the symbol definition, the linker cannot match up the names. Consider the following example:

```
main.cc
--------
int foo(int); // no "const" in declaration
int main()
{
    return foo(1);
}

file1.cc
---------
int foo(const int k) // "const" added to parameter declaration
{
    return k;
}

example% CC main.cc file1.cc
main.cc:
file1.cc:
Undefined                       first referenced
  symbol                           in file
int foo(int)                       main.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

You can see the reason for the failure by inspecting the names emitted by the compiler into the object files:

```
% nm main.o | grep foo
[2]    |        0|      0|NOTY |GLOB |0   |UNDEF |__1cDfoo6Fi_i_
% nm file1.o | grep foo
[2]     |     16|     40|FUNC |GLOB |0   |2    |__1cDfoo6Fki_i_
```

In `main.o`, the compiler emitted a reference to function `foo` that was mangled differently from the way the name was mangled in the definition of function `foo` in `file1.o`. As described in Section 1.7, “Anticipating Future Mangling Changes” on page 1-10, the workaround is not to use `const` in the declaration of `foo`’s parameter.

Programs that declare `foo` consistently with a `const` parameter, and programs that declare `foo` consistently with non-const parameter will compile and link successfully.

If we fixed the compiler bug, some programs that link now would stop linking. For example, suppose a 3rd-party binary library contained `file1.o`. If we fixed the compiler bug, no declaration of `foo` would allow a program to link to the `foo` in that library. If we do not fix the bug, you can declare `foo` with a `const` parameter and successfully link to the library.

Fortunately, all known compiler bugs related to name mangling result in "impossible" mangled names. That is, the invalid mangled name will never accidently refer to the mangled name of some other function or object. You can always add extra symbols to fix problems caused by incorrectly mangled names, as described elsewhere in the *Migration Guide.*

CHAPTER 2

Using Compatibility Mode

This chapter describes how to compile code that was intended for the C++ 4 compilers.

2.1 Compatibility Mode

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

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

For example:

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

There are some minor differences between using the C++ 4 compilers and the C++ 5 compilers in compatibility mode, as described in the following sections.

2.2 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.

2.3 Language Semantics

The C++ 5 compilers do a better job of enforcing some C++ language rules. They are 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 remainder of this section discusses the rules that previously were not enforced, but are now enforced by the C++ compiler.

Note – These rules are enforced by the C++ compiler in both compatibility mode and standard mode.

2.3.1 Copy Constructor

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
public:
  T();
};
T    f1(T t) { return t; }  // Error, can't return a T
void f2()    { f1( T() ); } // Error, can't pass a T
```

Solution: Make the copy constructor accessible. Usually, it is given public access.

2.3.2 Static Storage Class

The `static` storage class applies to objects and functions, 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.

2.3.3 Operators `new` and `delete`

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 the `delete` operator accessible. Usually, it is given public access.

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

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

2.3.4 new const

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
```

2.3.5 Conditional Expression

The C++ standard introduced a change in the rules for conditional expressions. The C++ compiler uses the new rule in both standard mode and compatibility mode. For more information, see Section 1.5, “Conditional Expressions” on page 1-8.

2.3.6 Default Parameter Value

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.

2.3.7 Trailing Commas

Trailing commas in function argument lists are not allowed.

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

Solution: Remove the extra comma.

2.3.8 Passing of `const` and Literal Values

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 Section 3.7, “String Literals and `char*`” on page 3-11 for related information.

2.3.9 Conversion Between Pointer-to-Function and `void*`

The C++ compiler, in both compatibility and standard mode, now issues a warning for implicit and explicit conversions between pointer-to-function and `void*`. For more information, see Section 1.6, “Function Pointers and `void*`” on page 1-9.

2.3.10 Type `enum`

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

```
enum E { zero=0, one=1 };
E foo(E e)
{
   e = 0;    // Error
   e = E(0); // OK
   return e;
}
```

Solution: Use a cast.

2.3.11 Member-Initializer List

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)
};
```

2.3.12 `const` and `volatile` Qualifiers

`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.

2.3.13 Nested Type

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
```

2.3.14 Class Template Definitions and Declarations

In class template definitions and declarations, appending the type argument bracketed by < > to the class's name has never been valid, but versions 4 and 5.0 of the C++ compiler did not report the error. For example, in the following code the <T> appended to `MyClass` is invalid for both the definition and the declaration.

```
template<class T> class MyClass<T> { ... }; // definition
template<class T> class MyClass<T>;         // declaration
```

Solution: Remove the bracketed type argument from the class name, as shown in the following code.

```
template<class T> class MyClass { ... }; // definition
template<class T> class MyClass;         // declaration
```

2.4 Template Compilation Model

The template compilation model for compatibility mode is different from the 4.2 compilation model. For more information about the new model, refer to Section 3.3.5, "Template Repository" on page 3-6.

CHAPTER 3

Using Standard Mode

This chapter explains use of the standard mode, which is the default compilation mode for the C++ compiler.

3.1 Standard Mode

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

```
-compat=5
```

For example:

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

3.2 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. (Determining when a programmer has used a keyword as an identifier is quite difficult, 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 are enabled or disabled as 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 below)

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`
`!`	`not`
`!=`	`not_eq`
`||`	`or`
`|`	`bitor`
`~`	`xor`
`~=`	`xor_eq`

3.3 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 C++ compiler does not enforce these rules, but it does recognize this keyword. In most cases, template code that worked under the 4.2 compiler 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.

3.3.1 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;
template< class T > class B { typedef int U; };
template< class T > class C : public B<T> {
  S s; // OK
  T t; // OK
  U x; // 1 No longer valid
  T::V z; // 2 No longer valid
};
```

The new language rules state that no base class that is dependent on a template parameter 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) 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 that must be a type.

The following modified example is correct.

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

3.3.2 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 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 C++ 4.1) that do not recognize `typename`, add `-DTYPENAME_NOT_RECOGNIZED` to the set of compiler options in your makefile.

3.3.3 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 C++ 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 compiler accepts the old syntax as an anachronism. The 4.2 compiler accepted the new specialization syntax, but it did not treat code using the new syntax correctly in every case. (The draft 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
```

3.3.4 Class Template Definitions and Declarations

In class template definitions and declarations, appending the type argument bracketed by < > to the class's name has never been valid, but versions 4 and 5.0 of the C++ compiler did not report the error. For example, in the following code the <T> appended to `MyClass` is invalid for both the definition and the declaration.

```
template<class T> class MyClass<T> { ... }; // definition
template<class T> class MyClass<T>;         // declaration
```

To resolve the problem, remove the bracketed type argument from the class name, as shown in the following code.

```
template<class T> class MyClass { ... }; // definition
template<class T> class MyClass;         // declaration
```

3.3.5 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 C++ 5 compilers, by default, use 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++ User's 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: 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 the *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.

3.3.6 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, the compiler might 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, the compiler might 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.

Here 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 do not 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 the *C++ Users' Guide* for more information.

3.4 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:

CODE EXAMPLE 3-1 Class Name Injection Problem 1

```
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.

CODE EXAMPLE 3-2 Class Name Injection Problem 2

```
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&);
```

3.5 `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++ 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++ 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
```

3.6 Conversion Between Pointer-to-Function and `void*`

The C++ compiler, in both compatibility and standard mode, now issues a warning for implicit and explicit conversions between pointer-to-function and `void*`. In standard mode, the compiler no longer recognizes such implicit conversions when resolving overloaded function calls. For more information, see Section 1.6, “Function Pointers and `void*`” on page 1-9.

3.7 String Literals and char*

Some history might help clarify 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] = 'G';
printf("%s", GREETING); // Prints "Gello 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 want 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 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.”)

In standard mode, the compiler 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 C++ compiler in compatibility mode puts the string literal in writable memory, just like the 4.2 compiler did. The program will run, although it technically has undefined behavior. In standard mode, the compiler 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.

3.8 Conditional Expressions

The C++ standard introduced a change in the rules for conditional expressions. The C++ compiler uses the new rule in both standard mode and compatibility mode. For more information, see Section 1.5, “Conditional Expressions” on page 1-8.

3.9 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 compatibility-mode run-time library (`libC`) depends on the old definitions and behavior, and the standard-mode run-time library (`libCstd`) 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
```

3.9.1 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.

3.9.2 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:

CODE EXAMPLE 3-3 Standard Header <new>

```
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.

3.9.3 Replacement Functions

If you have replacement versions of `operator new` and `delete`, they must match the signatures shown in CODE EXAMPLE 3-3, 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++ 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++ 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 standard mode, although the 4.2 compiler allowed it. You can, of course, write your own placement versions with different parameter lists.

3.9.4 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 Section 3.13, “Standard Exceptions” on page 3-23.

3.10 Boolean Type

The Boolean keywords—`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
```

You cannot define a Boolean type in compatibility mode that will work exactly like the new built-in `bool` type. This is one reason why a built-in Boolean type was added to C++.

3.11 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.

3.11.1 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.

In compatibility mode, the 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. This is the same behavior as the C++ 4.2 compiler.

In standard mode, the compiler 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. In compatibility mode the compiler accepts all combinations, just like the 4.2 compiler. In standard mode the compiler 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. 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.

3.11.2 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 because the calling sequence for C and C++ functions may be different on other systems.

3.11.3 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.

Good programming practice puts 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) { ... }
```

3.12 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 data size and in 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.)

3.13 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. In standard mode, the C++ 5 compilers use 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.

3.14 Order of the Destruction of Static Objects

A *static object* is an object with static storage duration. The static object can be global or in a namespace. It can be a static variable local to a function or it can be a static data member of a class.

The C++ standard requires that static objects be destroyed in the reverse order of their construction. In addition, the destruction of these objects might need to be intermixed with functions that are registered with the `atexit()` function.

Earlier versions of the C++ compiler destroyed the global static objects that are created in any one module in the reverse order of their construction. However, the correct destruction order over the entire program was not assured.

Beginning with version 5.1 of the C++ compiler, static objects are destroyed in strict reverse order of their construction. For example, suppose there are three static objects of type `T`:

- One object is at global scope in `file1`.
- A second object is at global scope in `file2`.
- The third object is at local scope in a function.

We can't predict which of the two global objects will be created first, the one in `file1` or the one in `file2`. However, the global object that is created first will be destroyed after the other global object is destroyed.

The local static object is created when its function is called. If the function is called after the creation of both the global static objects, the local object is destroyed before the global objects are destroyed.

The C++ standard places additional requirements on destruction of static objects in relation to functions registered with the `atexit()` function. If a function `F` is registered with `atexit()` after the construction of a static object `X`, `F` must be called at program exit before `X` is destroyed. Conversely, if function `F` is registered with `atexit()` before `X` is constructed, `F` must be called at program exit after `X` is destroyed.

Here is an example of this rule.

```
// T is a type having a destructor
void bar();
void foo()
{
  static T t2;
  atexit(bar);
  static T t3;
}
T t1;
int main()
{
  foo();
}
```

At program start, `t1` is created, then main runs. Main calls `foo()`. The `foo()` function performs the following in this order.

1. Create `t2`
2. Register `bar()` with `atexit()`
3. Create `t3`

Upon reaching the end of main, `exit` is called automatically. The sequence of the exit processing must be the following.

1. Destroy `t3`; `t3` was constructed after `bar()` was registered with `atexit()`
2. Run `bar()`
3. Destroy `t2`; `t2` was constructed before `bar()` was registered with `atexit()`

4. Destroy `t1`; `t1` was the first thing constructed, and therefore the last thing destroyed

Support for this interleaving of static destructors and the `atexit()` processing requires help from the Solaris run-time library `libc.so`. This support is available beginning with Solaris 8 software. A C++ program that is compiled with version 5.1, version 5.2, version 5.3, or version 5.4 of the C++ compiler looks, at runtime, for a special symbol in the library to determine whether it is currently running on a version of Solaris software that has this support. If the support is available, the static destructors are properly interleaved with `atexit`-registered functions. If the program is running on a version of Solaris software that does not have this support, the destructors are still executed in the proper order, but they are not interleaved with `atexit`-registered functions.

Notice that the determination is made by the program each time it runs. It does not matter what version of Solaris software you use to build the program. As long as the Solaris run-time library `libc.so` is linked dynamically (which happens by default), the interleaving at program exit will happen if the version of Solaris software that is running the program supports it.

Different compilers provide different levels of support for the correct order of the destruction of static objects. To improve the portability of your code, the correctness of your program should not depend on the exact order in which static objects are destroyed.

If your program depends on a particular order of destruction and worked with an older compiler, the order required by the standard might break the program in standard mode. The `-features=no%strictdestrorder` command option disables the strict ordering of destruction.

CHAPTER 4

Using Iostreams and Library Headers

This chapter explains the library and header file changes that were implemented in the C++ 5.0 compiler. You must consider these changes when migrating code that was intended for C++ 4 compilers for use with the C++ 5 compilers.

4.1 Iostreams

The C++ 4.2 compiler implemented *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.

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 when compiling in compatibility mode (`-compat[=4]`).

Note – Two versions of the classic iostream runtime library are supplied with the compiler. One version is compiled with the compiler in compatibility mode, and is the same as the library used with C++ 4.2. The other version is compiled from the same source code, but with the compiler in standard mode. The source-code interface is the same, but the binary code in the library has the standard-mode ABI. See Section 1.3, “Binary Compatibility Issues” on page 1-4.

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`.

Four of the standard headers are also provided in a form ending with ".h" that makes the header names available in the global namespace via using-declarations.

- <fstream.h>
- <iomanip.h>
- <iostream.h>
- <strstream.h>

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, CODE EXAMPLE 4-2 will compile with either classic iostreams or with the Sun implementation of standard iostreams.

CODE EXAMPLE 4-1 Using Standard `iostream` Name Forms

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

CODE EXAMPLE 4-2 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.

With classic iostreams, you can write your own forward declarations for iostream classes instead of including one of the iostream headers. For example:

CODE EXAMPLE 4-3 Forward Declaration With Classic iostreams

```
// valid for classic iostreams only
class istream;
class ostream;
class Myclass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

This approach will not work for standard iostreams, because classic names (`istream`, `ofstream`, `streambuf`, and so forth) are not the names of classes in standard iostreams. They are typedefs referring to specializations of class templates.

With standard iostreams, you cannot provide your own forward declarations of iostream classes. Instead, to provide correct forward declarations of the iostream classes, include the standard header `<iosfwd>`.

CODE EXAMPLE 4-4 Forward Declaration With Standard iostreams

```
// valid for standard iostreams only
#include <iosfwd>
using std::istream;
using std::ostream;
class MyClass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

To write code that will work with both standard and classic iostreams, you can include the full headers instead of using forward declarations. For example:

CODE EXAMPLE 4-5 Code for Both Classic and Standard iostreams

```
// valid for classic and standard iostreams with Sun C++
#include <iostream.h>
class MyClass;
istream& operator>>(istream&, MyClass&);
ostream& operator<<(ostream&, const MyClass&);
```

4.2 Task (Coroutine) Library

The coroutine library, accessed through 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.

4.3 Rogue Wave Tools.h++

The C++ compiler contains two versions of the Tools.h++ library:

- **One that works with classic iostreams.** This version of the Tools.h++ library is compatible with the Tools.h++ library that was shipped with earlier versions of the compiler.
  - **Standard Mode.** To use the classic iostreams version of Tools.h++ in standard mode (the default mode), use the `-library=rwtools7,iostream` option.
  - **Compatibility Mode.** To use the classic iostreams version of Tools.h++ in compatibility mode (`-compat[=4]`), use the `-library=rwtools7` option.
- **One that works with standard iostreams.** This version of the Tools.h++ library is incompatible with the classic iostreams version of Tools.h++. This version is available only in standard mode. It is not available in compatibility mode (`-compat[=4]`).

  To use the standard iostreams version of the library, use the `-library=rwtools7_std` option.

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

4.4 C Library Headers

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

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

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.

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 the Solaris 7 operating environment cannot be used directly. If you compile in standard mode, you must use the version of these headers that is supplied with the C++ compiler. If you use the wrong headers, your program can fail to compile or link.

With the Solaris 7 operating environment, you must 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
```

With the Solaris 8 operating environment, the standard C headers in `/usr/include` are correct for C++, and are used by the C++ compiler automatically. That is, if you write

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

you will get the C++ compiler's version of `stdio.h` when compiling on the Solaris 7 operating environment, but the Solaris version of `stdio.h` when compiling on the Solaris 8 operating environment. With the Solaris 8 operating environment, there is

no restriction against using the explicit path name in the include statement. However, use of path names, such as </usr/include/stdio.h>, does make the code unportable.

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 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.

Because 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>
extern FILE* f; // std::FILE would also be OK
int func1() { return fileno(f); }      // OK
int func2() { return std::fileno(f); } // Error
```

The following example uses <cstdio>:

```
#include <cstdio>
extern std::FILE* f; // FILE is only in namespace std
int func1() { return fileno(f); }      // Error
int func2() { return std::fileno(f); } // Error
```

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.

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>.

4.5 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/Cstd 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 `find` command, for example) you can look for the `.SUNWCCh` suffix.

CHAPTER 5

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 do not have to modify compiled C code to link it with C++ code. See “Commercially Available Books” on page -xvii for a list of books on the C++ language.

5.1 Reserved and Predefined Words

TABLE 5-1 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 5-1 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
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**

TABLE 5-1 Reserved Keywords *(Continued)*

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 C++ standard.

TABLE 5-2 C++ Reserved Words for Operators and Punctuators

and	bitor	not	or	xor
and_eq	compl	not_eq	or_eq	xor_eq
bitand				

5.2 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. For more information, see the *C++ Programming Guide*.

Note – Early C++ compilers pre-defined the macro `c_plusplus`, which is no longer supported. Use `__cplusplus` instead.

5.3 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.

5.4 Inlining Functions in Both C and C++

If an inline function definition is in source code that can be compiled by both the C compiler and the C++ compiler, then the function must comply with the following restrictions.

- The inline function declaration and definition must be enclosed by a conditional `extern "C"` statement as shown in the following example.

```
#ifdef __cplusplus
extern "C" {
#endif
inline int twice( int arg ) { return arg + arg; }
#ifdef __cplusplus
}
#endif
```

- The inline function's declaration and definition must meet the constraints imposed by both languages.
- The semantics of the function must be the same under both compilers. See appendix D of the C++ standard for program constructs that might yield different semantics in the two languages.

Index

T

V