C++ Programming Guide

Sun WorkShop 6

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 806-3571-10
May 2000, Revision A

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

Please
Recycle

Adobe PostScript

Important Note on New Product Names

As part of Sun's new developer product strategy, we have changed the names of our development tools from Sun WorkShop™ to Forte™ Developer products. The products, as you can see, are the same high-quality products you have come to expect from Sun; the only thing that has changed is the name.

We believe that the Forte™ name blends the traditional quality and focus of Sun's core programming tools with the multi-platform, business application deployment focus of the Forte tools, such as Forte Fusion™ and Forte™ for Java™. The new Forte organization delivers a complete array of tools for end-to-end application development and deployment.

For users of the Sun WorkShop tools, the following is a simple mapping of the old product names in WorkShop 5.0 to the new names in Forte Developer 6.

Old Product Name	New Product Name
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

In addition to the name changes, there have been major changes to two of the products.

- Forte for High Performance Computing contains all the tools formerly found in Sun Performance WorkShop Fortran and now includes the C++ compiler, so High Performance Computing users need to purchase only one product for all their development needs.
- Forte Fortran Desktop Edition is identical to the former Sun Performance WorkShop Personal Edition, except that the Fortran compilers in that product no longer support the creation of automatically parallelized or explicit, directive-based parallel code. This capability is still supported in the Fortran compilers in Forte for High Performance Computing.

We appreciate your continued use of our development products and hope that we can continue to fulfill your needs into the future.

Contents

Preface

This manual tells you how to use Sun WorkShop™ 6 C++ compiler features to write more efficient programs. This manual is intended for programmers with a working knowledge of C++ and some understanding of the Solaris™ operating environment and UNIX® commands.

Multiplatform Release

This Sun WorkShop release supports versions 2.6, 7, and 8 of the Solaris™ *SPARC™ Platform Edition* and Solaris *Intel Platform Edition* Operating Environments.

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.

Access to Sun WorkShop Development Tools

Because Sun WorkShop product components and man pages do not install into the standard `/usr/bin/` and `/usr/share/man` directories, you must change your `PATH` and `MANPATH` environment variables to enable access to Sun WorkShop compilers and tools.

To determine if you need to set your PATH environment variable:

1. **Display the current value of the PATH variable by typing:**

```
% echo $PATH
```

2. **Review the output for a string of paths containing /opt/SUNWspro/bin/.**

   If you find the paths, your PATH variable is already set to access Sun WorkShop development tools. If you do not find the paths, set your PATH environment variable by following the instructions in this section.

To determine if you need to set your MANPATH environment variable:

1. **Request the workshop man page by typing:**

```
% man workshop
```

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

   If the workshop(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 this section for setting your MANPATH environment variable.

Note – The information in this section assumes that your Sun WorkShop 6 products were installed in the /opt directory. If your Sun WorkShop software is not installed in the /opt directory, ask your system administrator for the equivalent path on your system.

The PATH and MANPATH variables should be set in your home .cshrc file if you are using the C shell or in your home .profile file if you are using the Bourne or Korn shells:

- To use Sun WorkShop commands, add the following to your PATH variable:

  /opt/SUNWspro/bin

- To access Sun WorkShop man pages with the man command, add the following to your MANPATH variable:

  /opt/SUNWspro/man

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 and MANPATH variables to access this release, see the *Sun WorkShop 6 Installation Guide* or your system administrator.

How This Book Is Organized

This book contains the following chapters:

Chapter 1, “Introduction,” briefly describes the features of the compiler.

Chapter 2, “Program Organization,” discusses header files, inline function definitions, and template definitions.

Chapter 3, “Pragmas,” provides information on using pragmas, or directives, to pass specific information to the compiler.

Chapter 4, “Templates,” discusses the definition and use of templates.

Chapter 5, “Exception Handling,” discusses the compiler’s implementation of exception handling.

Chapter 6, “Runtime Type Identification,” explains RTTI and introduces the RTTI options supported by the compiler.

Chapter 7, “Cast Operations,”describes new cast operations.

Chapter 8, “Performance,”explains how to improve the performance of C++ functions.

Chapter 9, “Multithreaded Programs,” explains how to build multithreaded programs. It also discusses the use of exceptions and explains how to share C++ Standard Library objects across threads.

Typographic Conventions

TABLE P-1 shows the typographic conventions that are used in Sun WorkShop documentation.

TABLE P-1 Typographic 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*.
`[ ]`	Square brackets contain arguments that are optional.	`-compat[=`*n*`]`
`( )`	Parentheses contain a set of choices for a required option.	`-d(y|n)`
`|`	The "pipe" or "bar" symbol separates arguments, only one of which may be used at one time.	`-d(y|n)`
...	The ellipsis indicates omission in a series.	`-features=`*a1*`[,...`*an*`]`
`%`	The percent sign indicates the word has a special meaning.	`-ftrap=%all,no%division`

Shell Prompts

TABLE P-2 shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P-2 Shell Prompts

Shell	Prompt
C shell	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Related Documentation

You can access documentation related to the subject matter of this book in the following ways:

- **Through the Internet at the `docs.sun.com`[sm] Web site.** You can search for a specific book title or you can browse by subject, document collection, or product at the following Web site:

  `http://docs.sun.com`

- **Through the installed Sun WorkShop products on your local system or network.** Sun WorkShop 6 HTML documents (manuals, online help, man pages, component readme files, and release notes) are available with your installed Sun WorkShop 6 products. To access the HTML documentation, do one of the following:
  - In any Sun WorkShop or Sun WorkShop™ TeamWare window, choose Help ➤ About Documentation.
  - In your Netscape™ Communicator 4.0 or compatible version browser, open the following file:

    `/opt/SUNWspro/docs/index.html`

  (If your Sun WorkShop software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open a document in the index, click the document's title.

Document Collections

TABLE P-3 lists related Sun WorkShop 6 manuals by document collection.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection

Document Collection	Document Title	Description
Forte™ Developer 6 / Sun WorkShop 6 Release Documents	*About Sun WorkShop 6 Documentation*	Describes the documentation available with this Sun WorkShop release and how to access it.
	What's New in Sun WorkShop 6	Provides information about the new features in the current and previous release of Sun WorkShop.
	Sun WorkShop 6 Release Notes	Contains installation details and other information that was not available until immediately before the final release of Sun WorkShop 6. This document complements the information that is available in the component readme files.
Forte Developer 6 / Sun WorkShop 6	*Analyzing Program Performance With Sun WorkShop 6*	Explains how to use the new Sampling Collector and Sampling Analyzer (with examples and a discussion of advanced profiling topics) and includes information about the command-line analysis tool `er_print`, the LoopTool and LoopReport utilities, and UNIX profiling tools `prof`, `gprof`, and `tcov`.
	Debugging a Program With dbx	Provides information on using dbx commands to debug a program with references to how the same debugging operations can be performed using the Sun WorkShop Debugging window.
	Introduction to Sun WorkShop	Acquaints you with the basic program development features of the Sun WorkShop integrated programming environment.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
Forte™ C 6 / Sun WorkShop 6 Compilers C	*C User's Guide*	Describes the C compiler options, Sun-specific capabilities such as pragmas, the `lint` tool, parallelization, migration to a 64-bit operating system, and ANSI/ISO-compliant C.
Forte™ C++ 6 / Sun WorkShop 6 Compilers C++	*C++ Library Reference*	Describes the C++ libraries, including C++ Standard Library, Tools.h++ class library, Sun WorkShop Memory Monitor, Iostream, and Complex.
	C++ Migration Guide	Provides guidance on migrating code to this version of the Sun WorkShop C++ compiler.
	C++ Programming Guide	Explains how to use the new features to write more efficient programs and covers templates, exception handling, runtime type identification, cast operations, performance, and multithreaded programs.
	C++ User's Guide	Provides information on command-line options and how to use the compiler.
	Sun WorkShop Memory Monitor User's Manual	Describes how the Sun WorkShop Memory Monitor solves the problems of memory management in C and C++. This manual is only available through your installed product (see `/opt/SUNWspro/docs/index.html`) and not at the `docs.sun.com` Web site.
Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	*Fortran Library Reference*	Provides details about the library routines supplied with the Fortran compiler.

TABLE P-3 Related Sun WorkShop 6 Documentation by Document Collection *(Continued)*

Document Collection	Document Title	Description
	Fortran Programming Guide	Discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
	Fortran User's Guide	Provides information on command-line options and how to use the compilers.
	FORTRAN 77 Language Reference	Provides a complete language reference.
	Interval Arithmetic Programming Reference	Describes the intrinsic `INTERVAL` data type supported by the Fortran 95 compiler.
Forte™ TeamWare 6 / Sun WorkShop TeamWare 6	*Sun WorkShop TeamWare 6 User's Guide*	Describes how to use the Sun WorkShop TeamWare code management tools.
Forte Developer 6/ Sun WorkShop Visual 6	*Sun WorkShop Visual User's Guide*	Describes how to use Visual to create C++ and Java™ graphical user interfaces.
Forte™ / Sun Performance Library 6	*Sun Performance Library Reference*	Discusses the optimized library of subroutines and functions used to perform computational linear algebra and fast Fourier transforms.
	Sun Performance Library User's Guide	Describes how to use the Sun-specific features of the Sun Performance Library, which is a collection of subroutines and functions used to solve linear algebra problems.
Numerical Computation Guide	*Numerical Computation Guide*	Describes issues regarding the numerical accuracy of floating-point computations.
Standard Library 2	*Standard C++ Class Library Reference*	Provides details on the Standard C++ Library.
	Standard C++ Library User's Guide	Describes how to use the Standard C++ Library.
Tools.h++ 7	*Tools.h++ Class Library Reference*	Provides details on the `Tools.h++` class library.
	Tools.h++ User's Guide	Discusses use of the C++ classes for enhancing the efficiency of your programs.

TABLE P-4 describes related Solaris documentation available through the `docs.sun.com` Web site.

TABLE P-4 Related Solaris Documentation

Document Collection	Document Title	Description
Solaris Software Developer	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker and the objects on which they operate.
	Programming Utilities Guide	Provides information for developers about the special built-in programming tools that are available in the Solaris operating environment.

Man Pages

The *C++ Library Reference* lists the man pages that are available for the C++ libraries. TABLE P-5 lists other man pages that are related to C++.

TABLE P-5 Man Pages Related to C++

Title	Description
`c++filt`	Copies each file name in sequence and writes it in the standard output after decoding symbols that look like C++ demangled names.
`dem`	Demangles one or more C++ names that you specify
`fbe`	Creates object files from assembly language source files.
`fpversion`	Prints information about the system CPU and FPU
`gprof`	Produces execution profile of a program
`ild`	Links incrementally, allowing insertion of modified object code into a previously built executable
`inline`	Expands assembler inline procedure calls
`lex`	Generates lexical analysis programs
`rpcgen`	Generates C/C++ code to implement an RPC protocol
`sigfpe`	Allows signal handling for specific SIGFPE codes
`stdarg`	Handles variable argument list

TABLE P-5 Man Pages Related to C++ *(Continued)*

Title	Description
`varargs`	Handles variable argument list
`version`	Displays version identification of object file or binary
`yacc`	Converts a context-free grammar into a set of tables for a simple automaton that executes an LALR(1) parsing algorithm

README File

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

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

To view the text version of the C++ compiler `README` file, type the following at a command prompt:

```
example% CC -xhelp=readme
```

To access the HTML version of the `README`, in your Netscape Communicator 4.0 or compatible version browser, open the following file:

`/opt/SUNWspro/docs/index.html`

(If your Sun WorkShop software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.) Your browser displays an index of Sun WorkShop 6 HTML documents. To open the `README`, find its entry in the index, then click the title.

Commercially Available Books

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

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).

CHAPTER 1

Introduction

The Sun WorkShop™ 6 C++ compiler, `CC`, described in this book (and the companion book, *C++ User's Guide*) is available under the Solaris 2.6, Solaris 7, and Solaris 8 operating environments on the SPARC™ and IA platforms. Sun WorkShop 6 C++ compiler implements the language and libraries described in the C++ International Standard.

1.1 The C++ Language

C++ was first described in *The C++ Programming Language* by Bjarne Stroustrup, and later more formally described in *The Annotated C++ Reference Manual*, by Margaret Ellis and Bjarne Stroustrup. An international standard for C++ is now available.

C++ is designed as a superset of the C programming language. While retaining efficient low-level programming, C++ adds:

- Stronger type checking
- Extensive data abstraction features
- Support for object-oriented programming
- Synchronous exception handling
- A large standard library

The support for object-oriented programming allows good design of modular and extensible interfaces among program modules. The standard library, including an extensible set of data types and algorithms, speeds the development of common applications.

1.1.1 Data Abstraction

C++ directly supports the use of programmer-defined data types that function much like the predefined data types already in the language. Such abstract data types can be defined to model the problem being solved.

1.1.2 Object-Oriented Features

The *class*, the fundamental unit of data abstraction in C++, contains data and defines operations on the data.

A class can build on one or more classes; this property is called *inheritance*, or *derivation*. The inherited class (or parent class) is called a *base* class in C++. It is known as a *super* class in other programming languages. The child class is called a *derived* class in C++. It is called a *subclass* in other programming languages. A derived class has all the data (and usually all the operations) of its base classes. It might add new data or replace operations from the base classes.

A class hierarchy can be designed to replace a base class with a derived class. For example, a `Window` class could have, as a derived class, a `ScrollingWindow` class that has all the properties of the `Window` class, but also allows scrolling of its contents. The `ScrollingWindow` class can then be used wherever the `Window` class is expected. This substitution property is known as polymorphism (meaning "many forms").

A program is said to be object-oriented when it is designed with abstract data types that use inheritance and exhibit polymorphism.

1.1.3 Type Checking

A compiler, or interpreter, performs *type checking* when it ensures that operations are applied to data of the correct type. C++ has stronger type checking than C, though not as strong as that provided by Pascal, which always prohibits attempts to use data of the wrong type. The C++ compiler produces errors in some cases, but in others, it converts data to the correct type.

In addition to having the C++ compiler perform these automatic conversions, you can explicitly convert between types using *type casts*.

A related area involves overloaded function names. In C++, you can give any number of functions the same name. The compiler decides which function should be called by checking the types of the parameters to the function call. If the correct function is not clear at compile time, the compiler issues an "ambiguity" error.

1.1.4 Classes and Data Abstraction

If you are a C programmer, think of a class as an extension of the `struct` type. A `struct` contains predefined data types, for example, `char` or `int`, and might also contain other `struct` types. C++ allows a `struct` type to have not only data types to store data, but also operations to manipulate the data. The C++ keyword `class` is analogous to `struct` in C. As a matter of style, many programmers use `struct` to mean a C-compatible `struct` type, and `class` to mean a `struct` type that has C++ features not available in C.

C++ provides classes as a means for *data abstraction*. You decide what types (classes) you want for your program data and then decide what operations each type needs. In other words, a C++ class is a user-defined data type.

For example, if you define a class `BigNum`, which implements arithmetic for very large integers, you can define the `+` operator so that it has a meaning when used with objects in the class `BigNum`. If, in the following expression, `n1` and `n2` are objects of the type `BigNum`, then the expression has a value determined by your definition of `+` for `BigNum`.

```
n1 + n2
```

In the absence of an `operator +()` that you define, the `+` operation would not be allowed on a class type. The `+` operator is predefined only for the built-in numeric types such as `int`, `long`, or `float`. Operators with such extra definitions are called *overloaded operators*.

The data storage elements in a C++ class are called *data members*. The operations in a C++ class include both functions and overloaded, built-in operators (special kinds of functions). A class's functions can be member functions (declared as part of the class), or nonmember functions (declared outside the class). Member functions exist to operate on members of the class. Nonmember functions must be declared *friend functions* if they need to access private or protected members of the class directly.

You can specify the level of access for a class member using the `public`, `private`, and `protected` *member access specifiers*. Public members are available to all functions in the program. Private members are available only to member functions and friend functions of the class. Protected members are available only to members and friends of the base class and members and friends of derived classes. You can apply the same access specifiers to base classes, limiting access to all members of the affected base class.

1.1.5 Compatibility With C

C++ was designed to be highly compatible with C. C programmers can learn C++ at their own pace and incorporate features of the new language when it seems appropriate. C++ supplements what is good and useful about C. Most important, C++ retains C's efficient interface to the hardware of the computer, including types and operators that correspond directly to components of computing equipment.

C++ does have some important differences. An ordinary C program might not be accepted by the C++ compiler without some modifications. See the *C++ Migration Guide* for information about what you must know to move from programming in C to programming in C++.

The differences between C and C++ are most evident in the way you can design interfaces between program modules, but C++ retains all of C's facilities for designing such interfaces. You can, for example, link C++ modules to C modules, so you can use C libraries with C++ programs.

C++ differs from C in a number of other details. In C++:

- Typed constants allow you to avoid the preprocessor and use named constants in your program.
- Function prototypes are required.
- The free store operators `new` and `delete` create dynamic objects of a specified type.
- References are automatically dereferenced pointers and act like alternative names for a variable. You can use references as function parameters.
- Special built-in operator names for type coercion are provided.
- Programmer-defined automatic type conversion is allowed.
- Variable declarations are allowed anywhere a statement may appear. They may also occur within the header of an `if`, `switch`, or `loop` statement, not just at the beginning of the block.
- A new comment marker begins a comment that extends to the end of the line.
- The name of an enumeration or class is automatically a type name.
- Default values can be assigned to function parameters.
- Inline functions can replace a function call with the function body, improving program efficiency without resorting to macros.

CHAPTER 2

Program Organization

The file organization of a C++ program requires more care than is typical for a C program. This chapter describes how to set up your header files, inline function definitions, and template definitions.

2.1 Header Files

Creating an effective header file can be difficult. Often your header file must adapt to different versions of both C and C++. To accommodate templates, make sure your header file is tolerant of multiple inclusions (idempotent), and is self-contained.

2.1.1 Language-Adaptable Header Files

You might need to develop header files for inclusion in both C and C++ programs. However, Kernighan and Ritchie C (K&R C), also known as "classic C," ANSI C, *Annotated Reference Manual* C++ (ARM C++), and ISO C++ sometimes require different declarations or definitions for the same program element within a single header file. (See the C++ *Migration Guide* for additional information on the variations between languages and versions.) To make header files acceptable to all these standards, you might need to use conditional compilation based on the existence or value of the preprocessor macros `__STDC__` and `__cplusplus`.

The macro __STDC__ is not defined in K&R C, but is defined in both ANSI C and C++. Use this macro to separate K&R C code from ANSI C or C++ code. This macro is most useful for separating prototyped from nonprototyped function definitions.

```
#ifdef _ _STDC_ _
int function(char*,...);      // C++ & ANSI C declaration
#else
int function();               // K&R C
#endif
```

The macro __cplusplus is not defined in C, but is defined in C++.

Note – Early versions of C++ defined the macro c_plusplus instead of _ _cplusplus. The macro c_plusplus is no longer defined.

Use the definition of the __cplusplus macro to separate C and C++. This macro is most useful in guarding the specification of an extern "C" interface for function declarations, as shown in the following example. To prevent inconsistent specification of extern "C", never place an #include directive within the scope of an extern "C" linkage specification.

```
#include "header.h"
...                     // ... other include files ...
#if defined(_ _cplusplus)
extern "C" {
#endif
  int g1();
  int g2();
  int g3()
#if defined(_ _cplusplus)
}
#endif
```

In ARM C++, the __cplusplus macro has a value of 1. In ISO C++, the macro has the value 199711L (the year and month of the standard expressed as a long constant). Use the value of this macro to separate ARM C++ from ISO C++. The macro value is most useful for guarding changes in template syntax.

```
// template function specialization
#if _ _cplusplus < 199711L
int power(int,int);                        // ARM C++
#else
template <> int power(int,int);            // ISO C++
#endif
```

2.1.2 Idempotent Header Files

Your header files should be idempotent. That is, the effect of including a header file many times should be exactly the same as including the header file only once. This property is especially important for templates. You can best accomplish idempotency by setting preprocessor conditions that prevent the body of your header file from appearing more than once.

```
#ifndef HEADER_H
#define HEADER_H
/* contents of header file */
#endif
```

2.1.3 Self-Contained Header Files

Your header files should include all the definitions that they need to be fully compilable. Make your header file self-contained by including within it all header files that contain needed definitions.

```
#include “another.h”
/* definitions that depend on another.h */
```

In general, your header files should be both idempotent and self-contained.

```
#ifndef HEADER_H
#define HEADER_H
#include “another.h”
/* definitions that depend on another.h */
#endif
```

2.1.4 Unnecessary Header File Inclusion

Programs written in C++ typically include many more declarations than do C programs, resulting in longer compilation times. You can reduce the number of declarations through judicious use of several techniques.

One technique is to conditionally include the header file itself, using the macro defined to make it idempotent. This approach introduces an additional interfile dependence.

```
#ifndef HEADER_H
#include “header.h”
#endif
```

Note – System header files often include guards of the form `_Xxxx`, where `X` is an uppercase letter. These identifiers are reserved and should *not* be used as a model for constructing macro guard identifiers.

Another way to reduce compilation time is to use incomplete class and structure declarations rather than including a header file that contains the definitions. This technique is applicable only if the complete definition is not needed, and if the identifier is actually a class or structure, and not a typedef or template. (The standard library has many typedefs that are actually templates and not classes.) For example, rather than writing:

```
#include “class.h”
a_class* a_ptr;
```

write:

```
class a_class;
a_class* a_ptr;
```

(If `a_class` is really a typedef, the technique does not work.)

One other technique is to use interface classes and factories, as described in the book *Design Patterns: Elements of Reusable Object-Oriented Software,* by Erich Gamma (Addison-Wesley, 1994).

2.2 Inline Function Definitions

You can organize your inline function definitions in two ways: with definitions inline and with definitions included. Each approach has advantages and disadvantages.

2.2.1 Function Definitions Inline

You can use the definitions-inline organization only with member functions. Place the body of the function directly following the function declaration within the class definition.

```
class Class
{
    int method() { return 3; }
};
```

This organization avoids repeating the prototype of the function, reduces the bulk of source files and the chance for inconsistencies. However, this organization can introduce implementation details into what would otherwise be read as an interface. You would have to do significant editing if the function became non-inline.

Use this organization only when the body of the function is trivial (that is, empty braces) or the function will always be inline.

2.2.2 Function Definitions Included

You can use the definitions-included organization for all inline functions. Place the body of the function together with a repeat (if necessary) of the prototype. The function definition may appear directly within the source file or be included with the source file.

```
class Class {
    int method();
};
inline int Class::method() {
    return 3;
}
```

This organization separates interface and implementation. You can move definitions easily from header files to source files when the function is no longer implemented inline. The disadvantage is that this organization repeats the prototype of the class, which increases the bulk of source files and the chance for inconsistencies.

2.3 Template Definitions

You can organize your template definitions in two ways: with definitions included and with definitions separated. The definitions-included organization allows greater control over template compilation.

2.3.1 Template Definitions Included

When you put the declarations and definitions for a template within the file that uses the template, the organization is *definitions-included*. For example:

main.cc

```
template <class Number> Number twice( Number original );
template <class Number> Number twice( Number original )
    { return original + original; }
int main( )
    { return twice<int>( -3 ); }
```

When a file using a template includes a file that contains both the template's declaration and the template's definition, the file that uses the template also has the definitions-included organization. For example:

twice.h

```
#ifndef TWICE_H
#define TWICE_H
template <class Number> Number twice( Number original );
template <class Number> Number twice( Number original )
    { return original + original; }
#endif
```

main.cc

```
#include “twice.h”
int main( )
    { return twice( -3 ); }
```

Note – It is very important to make your template headers idempotent. (See Section 2.1.2 “Idempotent Header Files.)

2.3.2 Template Definitions Separate

Another way to organize template definitions is to keep the definitions in template definition files, as shown in the following example.

twice.h	`template <class Number> Number twice( Number original );`
twice.cc	`template <class Number> Number twice( Number original )` `  { return original + original; }`

main.cc	`#include "twice.h"`
	`int main( )`
	`  { return twice<int>( -3 ); }`

Template definition files *must not* include any non-idempotent header files and often need not include any header files at all. (See Section 2.1.2 "Idempotent Header Files.)

Note – Although it is common to use source-file extensions for template definition files (`.c`, `.C`, `.cc`, `.cpp`, `.cxx`), template definition files are header files. The compiler includes them automatically if necessary. Template definition files should *not* be compiled independently.

If you place template declarations in one file and template definitions in another file, you have to be very careful how you construct the definition file, what you name it, and where you put it. You might also need to identify explicitly to the compiler the location of the definitions. Refer to *C++ User's Guide* for information about the template definition search rules.

CHAPTER 3

Pragmas

This chapter describes pragmas. A *pragma* is a compiler directive that allows you to provide additional information to the compiler. This information can change compilation details that are not otherwise under your control. For example, the `pack` pragma affects the layout of data within a structure. Compiler pragmas are also called *directives.*

The preprocessor keyword `pragma` is part of the C++ standard, but the form, content, and meaning of pragmas is different for every compiler. No pragmas are defined by the C++ standard. Code that depends on pragmas is not portable.

3.1 Pragma Forms

The various forms of a Sun WorkShop C++ compiler pragma are:

```
#pragma keyword
#pragma keyword ( a [ , a ] ...) [ , keyword ( a [ , a ] ...) ] ,...
#pragma sun keyword
```

The variable *keyword* identifies the specific directive; *a* indicates an argument.

The pragma keywords that are recognized by the Sun WorkShop C++ compiler are:

- `align`—Makes the parameter variables memory-aligned to a specified number of bytes, overriding the default.
- `init`—Marks a specified function as an initialization function.
- `fini`—Marks a specified function as a finalization function.
- `ident`—Places a specified string in the `.comment` section of the executable.

- `pack (`*n*`)`—Controls the layout of structure offsets. The value of *n* is a number—0, 1, 2, 4, or 8—that specifies the worst-case alignment desired for any structure member.
- `unknown_control_flow`—Specifies a list of routines that violate the usual control flow properties of procedure calls.
- `weak`—Defines weak symbol bindings.

3.2 Pragma Reference

This section describes the pragma keywords that are recognized by the Sun WorkShop C++ compiler.

3.2.1 `#pragma align`

`#pragma align` *integer*`(`*variable*`[,`*variable*`]...)`

Use `align` to make the listed variables memory-aligned to *integer* bytes, overriding the default. The following limitations apply:

- *integer* must be a power of 2 between 1 and 128; valid values are 1, 2, 4, 8, 16, 32, 64, and 128.
- *variable* is a global or static variable; it cannot be a local variable or a class member variable.
- If the specified alignment is smaller than the default, the default is used.
- The pragma line must appear before the declaration of the variables that it mentions; otherwise, it is ignored.
- Any variable mentioned on the pragma line but not declared in the code following the pragma line is ignored. Variables in the following example are properly declared.

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct S {int a; char *b;} astruct;
```

When #pragma align is used inside a namespace, mangled names must be used. For example, in the following code, the #pragma align statement will have no effect. To correct the problem, replace a, b, and c in the #pragma align statement with their mangled names.

```
namespace foo {
    #pragma align 8 (a, b, c)
    static char a;
    static char b;
    static char c;
}
```

3.2.2 #pragma init

#pragma init(*identifier* [, *identifier*] ...)

Use init to mark *identifier* as an initialization function. Such functions are expected to be of type void, to accept no arguments, and to be called while constructing the memory image of the program at the start of execution. Initializers in a shared object are executed during the operation that brings the shared object into memory, either at program start up or during some dynamic loading operation, such as dlopen(). The only ordering of calls to initialization functions is the order in which they are processed by the link editors, both static and dynamic.

Within a source file, the functions specified in #pragma init are executed after the static constructors in that file. You must declare the identifiers before using them in the pragma.

3.2.3 #pragma fini

#pragma fini (*identifier* [, *identifier*]...)

Use fini to mark *identifier* as a finalization function. Such functions are expected to be of type void, to accept no arguments, and to be called either when a program terminates under program control or when the containing shared object is removed from memory. As with initialization functions, finalization functions are executed in the order processed by the link editor.

In a source file, the functions specified in #pragma fini are executed after the static destructors in that file. You must declare the identifiers before using them in the pragma.

3.2.4 #pragma ident

`#pragma ident` *string*

Use `ident` to place *string* in the `.comment` section of the executable.

3.2.5 #pragma pack(*n*)

`#pragma pack([`*n*`])`

Use `pack` to affect the packing of structure members.

If present, *n* must be 0 or a power of 2. A value of other than 0 instructs the compiler to use the smaller of *n*-byte alignment and the platform's natural alignment for the data type. For example, the following directive causes the members of all structures defined after the directive (and before subsequent `pack` directives) to be aligned no more strictly than on 2-byte boundaries, even if the normal alignment would be on 4- or 8-byte boundaries.

```
#pragma pack(2)
```

When *n* is 0 or omitted, the member alignment reverts to the natural alignment values.

If the value of *n* is the same as or greater than the strictest alignment on the platform, the directive has the effect of natural alignment. The following table shows the strictest alignment for each platform.

TABLE 3-1 Strictest Alignment by Platform

Platform	Strictest Alignment
IA	4
SPARC generic, V7, V8, V8a, V8plus, V8plusa, V8plusb	8
SPARC V9, V9a, V9b	16

A `pack` directive applies to all structure definitions which follow it, until the next `pack` directive. If the same structure is defined in different translation units with different packing, your program may fail in unpredictable ways. In particular, you should not use a `pack` directive prior to including a header defining the interface of a precompiled library. The recommended usage is to place the `pack` directive in your program code, immediately before the structure to be packed, and to place `#pragma pack()` immediately after the structure.

When using `#pragma pack` on a SPARC platform to pack denser than the type's default alignment, the `-misalign` option must be specified for both the compilation and the linking of the application. The following table shows the storage sizes and default alignments of the integral data types.

TABLE 3-2 Storage Sizes and Default Alignments in Bytes

Type	SPARC V8 Size, Alignment	SPARC V9 Size, Alignment	IA Size, Alignment
bool	1, 1	1, 1	1, 1
char	1, 1	1, 1	1, 1
short	2, 2	2, 2	2, 2
wchar_t	4, 4	4, 4	4, 4
int	4, 4	4, 4	4, 4
long	4, 4	8, 8	4, 4
float	4, 4	4, 4	4, 4
double	8, 8	8, 8	8, 4
long double	16, 8	16, 16	12, 4
pointer to data	4, 4	8, 8	4, 4
pointer to function	4, 4	8, 8	4, 4
pointer to member data	4, 4	8, 8	4, 4
pointer to member function	8, 4	16, 8	8, 4

3.2.6 #pragma unknown_control_flow

`#pragma unknown_control_flow (`*name*`, [,`*name*`] ...)`

Use `unknown_control_flow` to specify a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`.

Because such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

3.2.7 #pragma weak

`#pragma weak` *name1* `[=` *name2*`]`

Use `weak` to define a weak global symbol. This pragma is used mainly in source files for building libraries. The linker does not warn you if it cannot resolve a weak symbol.

The `weak` pragma can specify symbols in one of two forms:

- **String form**. The string must be the mangled name for a C++ variable or function. The behavior for an invalid mangled name reference is unpredictable. The back end may or may not produce an error for invalid mangled name references. Regardless of whether it produces an error, the behavior of the back end when invalid mangled names are used is unpredictable.
- **Identifier form**. The identifier must be an unambiguous identifier for a C++ function that was previously declared in the compilation unit. The identifier form cannot be used for variables. The front end (`ccfe`) will produce an error message if it encounters an invalid identifier reference.

`#pragma weak` *name*

In the form `#pragma weak` *name*, the directive makes *name* a weak symbol. The linker will not complain if it does not find a symbol definition for *name*. It also does not complain about multiple weak definitions of the symbol. The linker simply takes the first one it encounters.

If another compilation unit has a strong definition for the function or variable, *name* will be linked to that. If there is no strong definition for *name*, the linker symbol will have a value of 0.

The following directive defines `ping` to be a weak symbol. No error messages are generated if the linker cannot find a definition for a symbol named `ping.`

```
#pragma weak ping
```

`#pragma weak` *name1* `=` *name2*

In the form #pragma weak *name1* = *name2*, the symbol *name1* becomes a weak reference to *name2*. If *name1* is not defined elsewhere, *name1* will have the value *name2*. If *name1* is defined elsewhere, the linker uses that definition and ignores the

weak reference to *name2*. The following directive instructs the linker to resolve any references to `bar` if it is defined anywhere in the program, and to `foo` otherwise.

```
#pragma weak bar = foo
```

In the identifier form, *name2* must be declared and defined within the current compilation unit. For example:

```
extern void bar(int) {...}
extern void _bar(int);
#pragma weak _bar=bar
```

When you use the string form, the symbol does not need to be previously declared. If both `_bar` and `bar` in the following example are `extern "C"`, the functions do not need to be declared. However, `bar` must be defined in the same object.

```
extern "C" void bar(int) {...}
#pragma weak "_bar" = "bar"
```

Overloading Functions

When you use the identifier form, there must be exactly one function with the specified name in scope at the pragma location. Attempting to use the identifier form of `#pragma weak` with an overloaded function is an error. For example:

```
int bar(int);
float bar(float);
#pragma weak bar        // error, ambiguous function name
```

To avoid the error, use the string form, as shown in the following example.

```
int bar(int);
float bar(float);
#pragma weak "__1cDbar6Fi_i_" // make float bar(int) weak
```

See the Solaris *Linker and Libraries Guide* for more information.

CHAPTER 4

Templates

Templates make it possible for you to write a single body of code that applies to a wide range of types in a type-safe manner. This chapter introduces template concepts and terminology in the context of function templates, discusses the more complicated (and more powerful) class templates, and describes the composition of templates. Also discussed are template instantiation, default template parameters, and template specialization. The chapter concludes with a discussion of potential problem areas for templates.

4.1 Function Templates

A function template describes a set of related functions that differ only by the types of their arguments or return values.

4.1.1 Function Template Declaration

You must declare a template before you can use it. A *declaration,* as in the following example, provides enough information to use the template, but not enough information to implement the template.

```
template <class Number> Number twice( Number original );
```

In this example, *Number* is a *template parameter;* it specifies the range of functions that the template describes. More specifically, *Number* is a *template type parameter,* and its use within the template definition stands for a type determined at the location where the template is used.

4.1.2 Function Template Definition

If you declare a template, you must also define it. A *definition* provides enough information to implement the template. The following example defines the template declared in the previous example.

```
template <class Number> Number twice( Number original )
    { return original + original; }
```

Because template definitions often appear in header files, a template definition might be repeated in several compilation units. All definitions, however, must be the same. This restriction is called the *One-Definition Rule*.

Sun WorkShop 6 C++ does not support expressions involving non-type template parameters in the function parameter list, as shown in the following example.

```
// Expressions with non-type template parameters
// in the function parameter list are not supported
template<int I> void foo( mytype<2*I> ) { ... }
template<int I, int J> void foo( int a[I+J] ) { ... }
```

4.1.3 Function Template Use

Once declared, templates can be used like any other function. Their *use* consists of naming the template and providing function arguments. The compiler can infer the template type arguments from the function argument types. For example, you can use the previously declared template as follows.

```
double twicedouble( double item )
    { return twice( item ); }
```

If a template argument cannot be inferred from the function argument types, it must be supplied where the function is called. For example:

```
template<class T> T func(); // no function arguments
int k = func<int>(); // template argument supplied explicitly
```

4.2 Class Templates

A class template describes a set of related classes or data types that differ only by types, by integral values, by pointers or references to variables with global linkage, or by a combination thereof. Class templates are particularly useful in describing generic, but type-safe, data structures.

4.2.1 Class Template Declaration

A class template declaration provides only the name of the class and its template arguments. Such a declaration is an *incomplete class template*.

The following example is a template declaration for a class named `Array` that takes any type as an argument.

```
template <class Elem> class Array;
```

This template is for a class named `String` that takes an `unsigned int` as an argument.

```
template <unsigned Size> class String;
```

4.2.2 Class Template Definition

A class template definition must declare the class data and function members, as in the following examples.

```
template <class Elem> class Array {
        Elem* data;
        int size;
    public:
        Array( int sz );
        int GetSize();
        Elem& operator[]( int idx );
};
```

```
template <unsigned Size> class String {
        char data[Size];
        static int overflows;
    public:
        String( char *initial );
        int length();
};
```

Unlike function templates, class templates can have both type parameters (such as `class Elem`) and expression parameters (such as `unsigned Size`). An expression parameter can be:

- A value that has an integral type or enumeration
- A pointer or a reference to an object
- A pointer or a reference to a function
- A pointer to a class member function

4.2.3 Class Template Member Definitions

The full definition of a class template requires definitions for its function members and static data members. Dynamic (nonstatic) data members are sufficiently defined by the class template declaration.

4.2.3.1 Function Member Definitions

The definition of a template function member consists of the template parameter specification followed by a function definition. The function identifier is qualified by the class template's class name and the template arguments. The following example shows definitions of two function members of the `Array` class template, which has a template parameter specification of `template <class Elem>`. Each function identifier is qualified by the template class name and the template argument `Array<Elem>`.

```
template <class Elem> Array<Elem>::Array( int sz )
    { size = sz; data = new Elem[ size ]; }

template <class Elem> int Array<Elem>::GetSize( )
    { return size; }
```

This example shows definitions of function members of the `String` class template.

```
#include <string.h>
template <unsigned Size> int String<Size>::length( )
    { int len = 0;
      while ( len < Size && data[len] != '\0' ) len++;
      return len; }

template <unsigned Size> String<Size>::String( char *initial )
    { strncpy( data, initial, Size );
      if ( length( ) == Size ) overflows++; }
```

4.2.3.2 Static Data Member Definitions

The definition of a template static data member consists of the template parameter specification followed by a variable definition, where the variable identifier is qualified by the class template name and its template actual arguments.

```
template <unsigned Size> int String<Size>::overflows = 0;
```

4.2.4 Class Template Use

A template class can be used wherever a type can be used. Specifying a template class consists of providing the values for the template name and arguments. The declaration in the following example creates the variable `int_array` based upon the `Array` template. The variable's class declaration and its set of methods are just like those in the `Array` template except that `Elem` is replaced with `int` (see Section 4.3 "Template Instantiation).

```
Array<int> int_array( 100 );
```

The declaration in this example creates the `short_string` variable using the `String` template.

```
String<8> short_string( "hello" );
```

You can use template class member functions as you would any other member function.

```
int x = int_array.GetSize( );
```

```
int x = short_string.length( );
```

4.3 Template Instantiation

Template *instantiation* involves generating a concrete class or function (*instance*) for a particular combination of template arguments. For example, the compiler generates a class for *Array<int>* and a different class for *Array<double>*. The new classes are defined by substituting the template arguments for the template parameters in the definition of the template class. In the *Array<int>* example, shown in the preceding "Class Templates" section, the compiler substitutes *int* wherever *Elem* appears.

4.3.1 Implicit Template Instantiation

The use of a template function or template class introduces the need for an instance. If that instance does not already exist, the compiler implicitly instantiates the template for that combination of template arguments.

4.3.2 Whole-Class Instantiation

When the compiler implicitly instantiates a template class, it usually instantiates only the members that are used. To force the compiler to instantiate all member functions when implicitly instantiating a class, use the `-template=wholeclass` compiler option. To turn this option off, specify the `-template=no%wholeclass` option, which is the default.

4.3.3 Explicit Template Instantiation

The compiler implicitly instantiates templates only for those combinations of template arguments that are actually used. This approach may be inappropriate for the construction of libraries that provide templates. C++ provides a facility to explicitly instantiate templates, as seen in the following examples.

4.3.3.1 Explicit Instantiation of Template Functions

To instantiate a template function explicitly, follow the `template` keyword by a declaration (not definition) for the function, with the function identifier followed by the template arguments.

```
template float twice<float>( float original );
```

Template arguments may be omitted when the compiler can infer them.

```
template int twice( int original );
```

4.3.3.2 Explicit Instantiation of Template Classes

To instantiate a template class explicitly, follow the `template` keyword by a declaration (not definition) for the class, with the class identifier followed by the template arguments.

```
template class Array<char>;
```

```
template class String<19>;
```

When you explicitly instantiate a class, all of its members are also instantiated.

4.3.3.3 Explicit Instantiation of Template Class Function Members

To explicitly instantiate a template class function member, follow the `template` keyword by a declaration (not definition) for the function, with the function identifier qualified by the template class, followed by the template arguments.

```
template int Array<char>::GetSize( );
```

```
template int String<19>::length( );
```

4.3.3.4 Explicit Instantiation of Template Class Static Data Members

To explicitly instantiate a template class static data member, follow the `template` keyword by a declaration (not definition) for the member, with the member identifier qualified by the template class, followed by the template argument.

```
template int String<19>::overflow;
```

4.4 Template Composition

You can use templates in a nested manner. This is particularly useful when defining generic functions over generic data structures, as in the standard C++ library. For example, a template sort function may be declared over a template array class:

```
template <class Elem> void sort( Array<Elem> );
```

and defined as:

```
template <class Elem> void sort( Array<Elem> store )
    { int num_elems = store.GetSize( );
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( store[j-1] > store[j] )
                  { Elem temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp; } }
```

The preceding example defines a sort function over the predeclared `Array` class template objects. The next example shows the actual use of the sort function.

```
Array<int> int_array( 100 );   // construct an array of ints
sort( int_array );             // sort it
```

4.5 Default Template Parameters

You can give default values to template parameters for class templates (but not function templates).

```
template <class Elem = int> class Array;
template <unsigned Size = 100> class String;
```

If a template parameter has a default value, all parameters after it must also have default values. A template parameter can have only one default value.

4.6 Template Specialization

There may be performance advantages to treating some combinations of template arguments as a special case, as in the following examples for `twice`. Alternatively, a template description might fail to work for a set of its possible arguments, as in the following examples for `sort`. Template specialization allows you to define alternative implementations for a given combination of actual template arguments. The template specialization overrides the default instantiation.

4.6.1 Template Specialization Declaration

You must declare a specialization before any use of that combination of template arguments. The following examples declare specialized implementations of *twice* and *sort*.

```
template <> unsigned twice<unsigned>( unsigned original );
```

```
template <> sort<char*>( Array<char*> store );
```

You can omit the template arguments if the compiler can unambiguously determine them. For example:

```
template <> unsigned twice( unsigned original );
```

```
template <> sort( Array<char*> store );
```

4.6.2 Template Specialization Definition

You must define all template specializations that you declare. The following examples define the functions declared in the preceding section.

```
template <> unsigned twice<unsigned>( unsigned original )
    { return original << 1; }
```

```
#include <string.h>
template <> void sort<char*>( Array<char*> store )
    { int num_elems = store.GetSize( );
      for ( int i = 0;  i < num_elems-1;  i++ )
          for ( int j = i+1;  j < num_elems;  j++ )
              if ( strcmp( store[j-1], store[j] ) > 0 )
                  { char *temp = store[j];
                    store[j] = store[j-1];
                    store[j-1] = temp; } }
```

4.6.3 Template Specialization Use and Instantiation

A specialization is used and instantiated just as any other template, except that the definition of a completely specialized template is also an instantiation.

4.6.4 Partial Specialization

In the previous examples, the templates are fully specialized. That is, they define an implementation for specific template arguments. A template can also be partially specialized, meaning that only some of the template parameters are specified, or that one or more parameters are limited to certain categories of type. The resulting partial specialization is itself still a template. For example, the following code sample shows a primary template and a full specializaton of that template.

```
template<class T, class U> class A { ... }; //primary template
template<> class A<int, double> { ... };    //specialization
```

The following code shows examples of partial specialization of the primary template.

```
template<classU> class A<int> { ... };          // Example 1
template<class T, class U> class A<T*> { ... }; // Example 2
template<class T> class A<T**, char> { ... };   // Example 3
```

- Example 1 provides a special template definition for cases when the first template parameter is type `int`.
- Example 2 provides a special template definition for cases when the first template parameter is any pointer type.
- Example 3 provides a special template definition for cases when the first template parameter is pointer-to-pointer of any type, and the second template parameter is type `char`.

4.7 Template Problem Areas

This section describes problems you might encounter when using templates.

4.7.1 Nonlocal Name Resolution and Instantiation

Sometimes a template definition uses names that are not defined by the template arguments or within the template itself. If so, the compiler resolves the name from the scope enclosing the template, which could be the context at the point of definition, or at the point of instantiation. A name can have different meanings in different places, yielding different resolutions.

Name resolution is complex. Consequently, you should not rely on nonlocal names, except those provided in a pervasive global environment. That is, use only nonlocal names that are declared and defined the same way everywhere. In the following example, the template function `converter` uses the nonlocal names `intermediary` and `temporary`. These names have different definitions in `use1.cc` and `use2.cc`, and will probably yield different results under different compilers. For templates to work reliably, all nonlocal names (`intermediary` and `temporary` in this case) must have the same definition everywhere.

use_common.h	// Common template definition template <class Source, class Target> Target converter(Source source) { temporary = (intermediary)source; return (Target)temporary; }
use1.cc	typedef int intermediary; int temporary; #include "use_common.h"
use2.cc	typedef double intermediary; unsigned int temporary; #include "use_common.h"

A common use of nonlocal names is the use of the `cin` and `cout` streams within a template. Few programmers really want to pass the stream as a template parameter, so they refer to a global variable. However, `cin` and `cout` must have the same definition everywhere.

4.7.2 Local Types as Template Arguments

The template instantiation system relies on type-name equivalence to determine which templates need to be instantiated or reinstantiated. Thus local types can cause serious problems when used as template arguments. Beware of creating similar problems in your code. For example:

CODE EXAMPLE 4-1 Example of Local Type as Template Argument Problem

array.h	`template <class Type> class Array {`
	`  Type* data;`
	`  int  size;`
	`public:`
	`  Array( int sz );`
	`  int GetSize( );`
	`};`

array.cc	`template <class Type> Array<Type>::Array( int sz )`
	`  { size = sz; data = new Type[size]; }`
	`template <class Type> int Array<Type>::GetSize( )`
	`  { return size;}`

file1.cc	`#include "array.h"`
	`struct Foo { int data; };`
	`Array<Foo> File1Data;`

file2.cc	`#include "array.h"`
	`struct Foo { double data; };`
	`Array<Foo> File2Data;`

The `Foo` type as registered in `file1.cc` is not the same as the `Foo` type registered in `file2.cc`. Using local types in this way could lead to errors and unexpected results.

4.7.3 Friend Declarations of Template Functions

Templates must be declared before they are used. A friend declaration constitutes a use of the template, not a declaration of the template. A true template declaration must precede the friend declaration. For example, when the compilation system attempts to link the produced object file for the following example, it generates an undefined error for the `operator<<` function, which is *not* instantiated.

CODE EXAMPLE 4-2 Example of Friend Declaration Problem

array.h

```
// generates undefined error for the operator<< function
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
        operator<<(std::ostream&, const array<T>&);
};
#endif
```

array.cc

```
#include <stdlib.h>
#include <iostream>

template<class T> array<T>::array() { size = 1024; }

template<class T>
std::ostream&
operator<<(std::ostream& out, const array<T>& rhs)
    { return out << '[' << rhs.size << ']'; }
```

main.cc

```
#include <iostream>
#include "array.h"

int main()
{
    std::cout
      << "creating an array of int... " << std::flush;
    array<int> foo;
    std::cout << "done\n";
    std::cout << foo << std::endl;
    return 0;
}
```

Note that there is no error message during compilation because the compiler reads the following as the declaration of a normal function that is a `friend` of the `array` class.

```
friend ostream& operator<<(ostream&, const array<T>&);
```

Because `operator<<` is really a template function, you need to supply a template declaration for prior to the declaration of `template class array`. However, because `operator<<` has a parameter of `type array<T>`, you must precede the function declaration with a declaration of `array<T>`. The file `array.h` must look like this:

```
#ifndef ARRAY_H
#define ARRAY_H
#include <iosfwd>

// the next two lines declare operator<< as a template function
template<class T> class array;
template<class T>
std::ostream& operator<<(std::ostream&, const array<T>&);

template<class T> class array {
    int size;
public:
    array();
    friend std::ostream&
      operator<<(std::ostream&, const array<T>&);
};
#endif
```

4.7.4 Using Qualified Names Within Template Definitions

The C++ standard requires types with qualified names that depend upon template arguments to be explicitly noted as type names with the `typename` keyword. This is true even if the compiler can "know" that it should be a type. The comments in the following example show the types with qualified names that require the `typename` keyword.

```
struct simple {
  typedef int a_type;
  static int a_datum;
};
int simple::a_datum = 0; // not a type
template <class T> struct parametric {
  typedef T a_type;
  static T a_datum;
};
template <class T> T parametric<T>::a_datum = 0;   // not a type
template <class T> struct example {
  static typename T::a_type variable1;             // dependent
  static typename parametric<T>::a_type variable2; // dependent
  static simple::a_type variable3;               // not dependent
};
template <class T> typename T::a_type             // dependent
  example<T>::variable1 = 0;                      // not a type
template <class T> typename parametric<T>::a_type // dependent
  example<T>::variable2 = 0;                      // not a type
template <class T> simple::a_type   // not dependent
example<T>::variable3 = 0;          // not a type
```

4.7.5 Nesting Template Declarations

Because the ">>" character sequence is interpreted as the right-shift operator, you must be careful when you use one template declaration inside another. Make sure you separate adjacent ">" characters with at least one blank space.

For example, the following ill-formed statement:

```
// ill-formed statement
Array<String<10>> short_string_array(100); // >> = right-shift
```

is interpreted as:

```
Array<String<10  >> short_string_array(100);
```

The correct syntax is:

```
Array<String<10> > short_string_array(100);
```

CHAPTER 5

Exception Handling

This chapter explains exception handling as it is currently implemented in the Sun C++ compiler and discusses the requirements of the C++ International Standard.

For additional information on exception handling, see *The C++ Programming Language,* Third Edition, by Bjarne Stroustrup (Addison-Wesley, 1997).

5.1 Understanding Exception Handling

Exceptions are anomalies that occur during the normal flow of a program and prevent it from continuing. These anomalies—user, logic, or system errors—can be detected by a function. If the detecting function cannot deal with the anomaly, it "throws" an exception. A function that "handles" that kind of exception catches it.

In C++, when an exception is thrown, it cannot be ignored—there must be some kind of notification or termination of the program. If no user-provided exception handler is present, the compiler provides a default mechanism to terminate the program.

Exception handling is expensive compared to ordinary program flow controls, such as loops or if-statements. It is therefore better not to use the exception mechanism to deal with ordinary situations, but to reserve it for situations that are truly unusual.

Exceptions are particularly helpful in dealing with situations that cannot be handled locally. Instead of propagating error status throughout the program, you can transfer control directly to the point where the error can be handled.

For example, a function might have the job of opening a file and initializing some associated data. If the file cannot be opened or is corrupted, the function cannot do its job. However, that function might not have enough information to handle the problem. The function can throw an exception object that describes the problem, transferring control to an earlier point in the program. The exception handler might

automatically try a backup file, query the user for another file to try, or shut down the program gracefully. Without exception handlers, status and data would have to be passed down and up the function call hierarchy, with status checks after every function call. With exception handlers, the flow of control is not obscured by error checking. If a function returns, the caller can be certain that it succeeded.

Exception handlers have disadvantages. If a function does not return because it, or some other function it called, threw an exception, data might be left in an inconsistent state. You need to know when an exception might be thrown, and whether the exception might have a bad effect on the program state.

For information about using exceptions in a multithreaded environment, see Section 9.2 "Using Exceptions in a Multithreaded Program" on page 9-3.

5.2 Using Exception Handling Keywords

There are three keywords for exception handling in C++:

- `try`
- `catch`
- `throw`

5.2.1 `try`

A `try` block is a group of C++ statements, enclosed in braces `{ }`, that might cause an exception. This grouping restricts exception handlers to exceptions generated within the `try` block. Each `try` block has one or more associated `catch` blocks.

5.2.2 `catch`

A `catch` block is a group of C++ statements that are used to handle a specific thrown exception. One or more `catch` blocks, or *handlers*, should be placed after each `try` block. A `catch` block is specified by:

1. The keyword `catch`
2. A `catch` parameter, enclosed in parentheses `()`, which corresponds to a specific type of exception that may be thrown by the `try` block
3. A group of statements, enclosed in braces `{ },` whose purpose is to handle the exception

5.2.3 `throw`

The `throw` statement is used to throw an exception and its value to a subsequent exception handler. A regular throw consists of the keyword `throw` and an expression. The result type of the expression determines which `catch` block receives control. Within a `catch` block, the current exception and value may be re-thrown simply by specifying the `throw` keyword alone (with no expression).

In the following example, the function call in the `try` block passes control to `f()`, which throws an exception of type `Overflow`. This exception is handled by the `catch` block, which handles type `Overflow` exceptions.

```
class Overflow {
                        // ...
public:
   Overflow(char,double,double);
};

void f(double x)
{
                        // ...
   throw Overflow('+',x,3.45e107);
}

int main() {
    try {
                        // ...
        f(1.2);
                        //...
    }
    catch(Overflow& oo) {
                        // handle exceptions of type Overflow here
    }
}
```

5.3 Implementing Exception Handlers

To implement an exception handler, perform these basic tasks:

- When a function is called by many other functions, code it so that an exception is thrown whenever an error is detected. The `throw` expression throws an object. This object is used to identify the types of exceptions and to pass specific information about the exception that has been thrown.

- Use the `try` statement in a client program to anticipate exceptions. Enclose function calls that might produce an exception in a `try` block.
- Code one or more `catch` blocks immediately after the `try` block. Each `catch` block identifies what type or class of objects it is capable of catching. When an object is thrown by the exception, the following actions occur:
  - If the object thrown by the exception matches the type of the `catch` expression, control passes to that `catch` block.
  - If the object thrown by the exception does not match the first `catch` block, subsequent `catch` blocks are searched for a matching type (see Section 5.8 "Matching Exceptions With Handlers").
  - If there is no catch block at the current scope matching the thrown exception, the current scope is exited, and all automatic (local nonstatic) objects defined in that scope are destroyed. The surrounding scope (which might be function scope) is checked for a matching handler. This process is continued until a scope is found that has a matching catch block. If one is found before exiting function `main()`, that catch block is entered.
  - If there is no match in any of the `catch` blocks, the program is normally terminated with a call to the predefined function `terminate()`. By default, `terminate()` calls `abort()`, which destroys all remaining objects and exits from the program. This default behavior can be changed by calling the `set_terminate()` function.

5.3.1 Synchronous Exception Handling

Exception handling is designed to support only synchronous exceptions, such as array range checks. The term *synchronous exception* means that exceptions can only be originated from `throw` expressions.

The C++ standard supports synchronous exception handling with a termination model. *Termination* means that once an exception is thrown, control never returns to the throw point.

5.3.2 Asynchronous Exception Handling

Exception handling is not designed to directly handle asynchronous exceptions such as keyboard interrupts. However, you can make exception handling work in the presence of asynchronous events if you are careful. For instance, to make exception handling work with signals, you can write a signal handler that sets a global variable, and create another routine that polls the value of that variable at regular intervals and throws an exception when the value changes. You cannot throw an exception from a signal handler.

5.4 Managing Flow of Control

In C++, exception handlers do not correct the exception and then return to the point at which the exception occurred. Instead, when an exception is generated, control is passed out of the block that threw the exception, out of the `try` block that anticipated the exception, and into the `catch` block whose exception declaration matches the exception thrown.

The `catch` block handles the exception. It might rethrow the same exception, throw another exception, jump to a label, return from the function, or end normally. If a `catch` block ends normally, without a `throw`, the flow of control passes over all other `catch` blocks associated with the `try` block.

Whenever an exception is thrown and caught, and control is returned outside of the function that threw the exception, *stack unwinding* takes place. During stack unwinding, any automatic objects that were created within the scope of the block that was exited are safely destroyed via calls to their destructors.

If a `try` block ends without an exception, all associated `catch` blocks are ignored.

Note – An exception handler cannot return control to the source of the error by using the `return` statement. A `return` statement issued in this context returns from the function containing the `catch` block.

5.4.1 Branching Into and Out of `try` Blocks and Handlers

Branching out of a `try` block or a handler is allowed. Branching into a `catch` block is not allowed, however, because that is equivalent to jumping past an initiation of the exception.

5.4.2 Nesting of Exceptions

Nesting of exceptions, that is, throwing an exception while another remains unhandled, is allowed only in restricted circumstances. From the point when an exception is thrown to the point when the matching `catch` clause is entered, the exception is unhandled. Functions that are called along the way, such as destructors of automatic objects being destroyed, may throw new exceptions, as long as the

exception does not escape the function. If a function exits via an exception while another exception remains unhandled, the `terminate()` function is called immediately.

Once an exception handler has been entered, the exception is considered handled, and exceptions may be thrown again.

You can determine whether any exception has been thrown and is currently unhandled. See Section 5.7 "Calling the `uncaught_exception()` Function" on page 5-10.

5.4.3 Specifying Exceptions to Be Thrown

A function declaration can include an *exception specification*, a list of exceptions that a function may throw, directly or indirectly.

The two following declarations indicate to the caller that the function `f1` generates only exceptions that can be caught by a handler of type `X`, and that the function `f2` generates only exceptions that can be caught by handlers of type `W`, `Y`, or `Z`:

```
void f1(int) throw(X);
void f2(int) throw(W,Y,Z);
```

A variation on the previous example is:

```
void f3(int) throw(); // empty parentheses
```

This declaration guarantees that no exception is generated by the function `f3`. If a function exits through any exception that is not allowed by an exception specification, it results in a call to the predefined function `unexpected()`. By default, `unexpected()` calls `terminate()` which by default exits the program. You can change this default behavior by calling the `set_unexpected()` function. See Section 5.6.2 "`set_unexpected()`" on page 5-9.

The check for unexpected exceptions is done at program execution time, not at compile time. Even if it appears that a disallowed exception might be thrown, there is no error unless the disallowed exception is actually thrown at runtime.

The compiler can, however, eliminate unnecessary checking in some simple cases. For instance, no checking for `f` is generated in the following example.

```
void foo(int) throw(x);
void f(int) throw(x);
{   foo(13);
}
```

The absence of an exception specification allows any exception to be thrown.

5.5 Specifying Runtime Errors

There are five runtime error messages associated with exceptions:

- No handler for the exception
- Unexpected exception thrown
- An exception can only be re-thrown in a handler
- During stack unwinding, a destructor must handle its own exception
- Out of memory

When errors are detected at runtime, the error message displays the type of the current exception and one of the five error messages. By default, the predefined function `terminate()` is called, which then calls `abort()`.

The compiler uses the information provided in the exception specification to optimize code production. For example, table entries for functions that do not throw exceptions are suppressed, and runtime checking for exception specifications of functions is eliminated wherever possible. Thus, declaring functions with correct exception specifications can lead to better code generation.

5.6 Modifying the `terminate()` and `unexpected()` Functions

The following sections describe how to modify the behavior of the `terminate()` and `unexpected()` functions using `set_terminate()` and `set_unexpected()`. For information about using these functions in a multithreaded environment, see Section 9.2 “Using Exceptions in a Multithreaded Program” on page 9-3.

5.6.1 set_terminate()

You can modify the default behavior of `terminate()` by calling the function `set_terminate()`, as shown in the following example.

```
// declarations are in standard header <exception>
namespace std {
   typedef void (*terminate_handler)();
   terminate_handler set_terminate(terminate_handler f) throw();
   void terminate();
}
```

The `terminate()` function is called in any of the following circumstances:

- The exception handling mechanism calls a user function (including destructors for automatic objects) that exits through an uncaught exception while another exception remains uncaught.
- The exception handling mechanism cannot find a handler for a thrown exception.
- The construction or destruction of a nonlocal object with static storage duration exits using an exception.
- Execution of a function registered with `atexit()` exits using an exception.
- A `throw` expression with no operand attempts to rethrow an exception and no exception is being handled.
- The `unexpected()` function throws an exception that is not allowed by the previously violated exception specification, and `std::bad_exception` is not included in that exception specification.
- The default version of `unexpected()` is called.

The `terminate()` function calls the function passed as an argument to `set_terminate()`. Such a function takes no parameters, returns no value, and must terminate the program (or the current thread). The function passed in the most recent call to `set_terminate()` is called. The previous function passed as an argument to `set_terminate()` is the return value, so you can implement a stack strategy for using `terminate()`. The default function for `terminate()` calls `abort()` for the main thread and `thr_exit()` for other threads. Note that `thr_exit()` does not unwind the stack or call C++ destructors for automatic objects.

Note – A replacement for `terminate()` must not return to its caller.

5.6.2 set_unexpected()

You can modify the default behavior of `unexpected()` by calling the function `set_unexpected()`, as shown in the following example.

```
// declarations are in standard header <exception>
namespace std {
   class exception;
   class bad_exception;
   typedef void (*unexpected_handler)();
   unexpected_handler
      set_unexpected(unexpected_handler f) throw();
   void unexpected();
}
```

The `unexpected()` function is called when a function attempts to exit through an exception not listed in its exception specification. The default version of `unexpected()` calls `terminate()`.

A replacement version of `unexpected()` might throw an exception permitted by the violated exception specification. If it does so, exception handling continues as though the original function had really thrown the replacement exception. If the replacement for `unexpected()` throws any other exception, that exception is replaced by the standard exception `std::bad_exception`. If the original function's exception specification does not allow `std::bad_exception`, function `terminate()` is called immediately. Otherwise, exception handling continues as though the original function had really thrown `std::bad_exception`.

`unexpected()` calls the function passed as an argument to `set_unexpected()`. Such a function takes no parameters, returns no value, and must not return to its caller. The function passed in the most recent call to `set_unexpected()` is called. The previous function passed as an argument to `set_unexpected()` is the return value, so you can implement a stack strategy for using `unexpected()`.

Note – A replacement for `unexpected()` must not return to its caller.

5.7 Calling the `uncaught_exception()` Function

An uncaught, or active, exception is an exception that has been thrown, but not yet accepted by a handler. The function `uncaught_exception()` returns `true` if there is an uncaught exception, and `false` otherwise.

The `uncaught_exception()` function is most useful for preventing program termination due to a function that exits with an uncaught exception while another exception is still active. This situation most commonly occurs when a destructor called during stack unwinding throws an exception. To prevent this situation, make sure `uncaught_exception()` returns `false` before throwing an exception within a destructor. (Another way to prevent such termination is to design your program so that destructors do not need to throw exceptions.)

5.8 Matching Exceptions With Handlers

A handler type `T` matches a `throw` type `E` if any one of the following is true:

- `T` is the same as `E`.
- `T` is `const` or `volatile` of `E`.
- `E` is `const` or `volatile` of `T`.
- `T` is ref of `E` or `E` is ref of `T`.
- `T` is a `public` base class of `E`.
- `T` and `E` are both pointer types, and `E` can be converted to `T` by a standard pointer conversion.

Throwing exceptions of reference or pointer types can result in a dangling pointer if the object pointed or referred to is destroyed before exception handling is complete. When an object is thrown, a copy of the object is always made through the copy constructor, and the copy is passed to the `catch` block. It is therefore safe to throw a local or temporary object.

While handlers of type `(X)` and `(X&)` both match an exception of type `X`, the semantics are different. Using a handler with type (`X`) invokes the object's copy constructor (again). If the thrown object is of a type derived from the handler type, the object is truncated. Catching a class object by reference therefore usually executes faster.

Handlers for a `try` block are tried in the order of their appearance. Handlers for a derived class (or a pointer to a reference to a derived class) must precede handlers for the base class to ensure that the handler for the derived class can be invoked.

5.9 Checking Access Control in Exceptions

The compiler performs the following check on access control for exceptions:

- The formal argument of a `catch` clause obeys the same rules as an argument of the function in which the `catch` clause occurs.
- An object can be thrown if it can be copied and destroyed in the context of the function in which the `throw` occurs.

Currently, access controls do not affect matching.

No other access is checked at runtime except for the matching rule described in Section 5.8 “Matching Exceptions With Handlers.

5.10 Enclosing Functions in `try` Blocks

If the constructor for a base class or member of a class `T` exits via an exception, there would ordinarily be no way for the `T` constructor to detect or handle the exception. The exception would be thrown before the body of the `T` constructor is entered, and thus before any `try` block in `T` could be entered.

A new feature in C++ is the ability to enclose an entire function in a `try` block. For ordinary functions, the effect is no different from placing the body of the function in a `try` block. But for a constructor, the `try` block traps any exceptions that escape from initializers of base classes and members of the constructor’s class. When the entire function is enclosed in a `try` block, the block is called a *function try block*.

In the following example, any exception thrown from the constructor of base class `B` or member `e` is caught before the body of the `T` constructor is entered, and is handled by the matching `catch` block.

You cannot use a `return` statement in the handler of a function `try` block, because the `catch` block is outside the function. You can only throw an exception or terminate the program by calling `exit()` or `terminate()`.

```
class B { ... };
class E { ... };
class T : public B {
public:
    T();
private:
    E e;
};
T::T()
try  : B(args), e(args)
{
    ... // body of constructor
}
catch( X& x ) {
   ... // handle exception X
}
catch( ... ) {
   ... // handle any other exception
}
```

5.11 Disabling Exceptions

If you know that exceptions are not used in a program, you can use the compiler option `-features=no%except` to suppress generation of code that supports exception handling. The use of the option results in slightly smaller code size and faster code execution. However, when files compiled with exceptions disabled are linked to files using exceptions, some local objects in the files compiled with exceptions disabled are not destroyed when exceptions occur. By default, the compiler generates code to support exception handling. Unless the time and space overhead is important, it is usually better to leave exceptions enabled.

5.12 Using Runtime Functions and Predefined Exceptions

The standard header `<exception>` provides the classes and exception-related functions specified in the C++ standard. You can access this header only when compiling in standard mode (compiler default mode, or with option `-compat=5`). The following excerpt shows the `<exception>` header file declarations.

```
// standard header <exception>
namespace std {
    class exception {
        exception() throw();
        exception(const exception&) throw();
        exception& operator=(const exception&) throw();
        virtual ~exception() throw();
        virtual const char* what() const throw();
    };
    class bad_exception: public exception { ... };
    // Unexpected exception handling
    typedef void (*unexpected_handler)();
    unexpected_handler
        set_unexpected(unexpected_handler) throw();
    void unexpected();
    // Termination handling
    typedef void (*terminate_handler)();
    terminate_handler set_terminate(terminate_handler) throw();
    void terminate();
    bool uncaught_exception() throw();
}
```

The standard class `exception` is the base class for all exceptions thrown by selected language constructs or by the C++ standard library. An object of type `exception` can be constructed, copied, and destroyed without generating an exception. The virtual member function `what()` returns a character string that describes the exception.

For compatibility with exceptions as used in C++ release 4.2, the header <exception.h> is also provided for use in standard mode. This header allows for a transition to standard C++ code and contains declarations that are not part of standard C++. Update your code to follow the C++ standard (using <exception> instead of <exception.h>) as development schedules permit.

```
// header <exception.h>, used for transition
#include <exception>
#include <new>
using std::exception;
using std::bad_exception;
using std::set_unexpected;
using std::unexpected;
using std::set_terminate;
using std::terminate;
typedef std::exception xmsg;
typedef std::bad_exception xunexpected;
typedef std::bad_alloc xalloc;
```

In compatibility mode (–compat[=4]), header <exception> is not available, and header <exception.h> refers to the same header provided with C++ release 4.2. It is not reproduced here.

5.13 Mixing Exceptions With Signals and Setjmp/Longjmp

You can use setjmp/longjmp in a program where exceptions can occur, as long as they don't interact.

All the rules for using exceptions and setjmp/longjmp separately apply. In addition, a longjmp from point A to point B is valid only if an exception thrown at A and caught at B would have the same effect. In particular, you must not longjmp into or out of a try-block or catch-block (directly or indirectly), or longjmp past the initialization or non-trivial destruction of auto or temporary variables.

You cannot throw an exception from a signal handler.

5.14 Building Shared Libraries That Have Exceptions

When shared libraries are opened with `dlopen,` you must use `RTLD_GLOBAL` for exceptions to work.

Note – When building shared libraries that contain exceptions, do not pass the option `-Bsymbolic` to `ld`. Exceptions that should be caught might be missed.

CHAPTER 6

Runtime Type Identification

This chapter explains the use of Runtime Type Identification (RTTI). Use this feature while a program is running to find out type information that you could not determine at compile time.

6.1 Static and Dynamic Types

In C++, pointers to classes have a *static* type, the type written in the pointer declaration, and a *dynamic* type, which is determined by the actual type referenced. The dynamic type of the object could be any class type derived from the static type. In the following example, `ap` has the static type `A*` and a dynamic type `B*`.

```
class A {};
class B: public A {};
extern B bv;
extern A* ap = &bv;
```

RTTI allows the programmer to determine the dynamic type of the pointer.

6.2 RTTI Options

In compatibility mode (`-compat[=4]`), RTTI support requires significant resources to implement. RTTI is disabled by default in that mode. To enable RTTI implementation and recognition of the associated `typeid` keyword, use the option `-features=rtti`. To disable RTTI implementation and recognition of the associated `typeid` keyword, use the option `–features=no%rtti` (the default).

In standard mode (the default mode), RTTI does not have a significant impact on program compilation or execution. RTTI is always enabled in standard mode.

6.3 `typeid` Operator

The `typeid` operator produces a reference to an object of class `type_info`, which describes the most-derived type of the object. To make use of the `typeid()` function, the source code must `#include` the `<typeinfo>` header file. The primary value of this operator and class combination is in comparisons. In such comparisons, the top-level `const` and `volatile` qualifiers are ignored, as in the following example. Note that, in this example, `A` and `B` are types which have default constructors.

```
#include <typeinfo>
#include <assert.h>
void use_of_typeinfo( )
{
      A a1;
      const A a2;
      assert( typeid(a1) == typeid(a2) );
      assert( typeid(A)  == typeid(const A) );
      assert( typeid(A)  == typeid(a2) );
      assert( typeid(A)  == typeid(const A&) );
      B b1;
      assert( typeid(a1) != typeid(b1) );
      assert( typeid(A)  != typeid(B) );
}
```

The `typeid` operator throws a `bad_typeid` exception when given a null pointer.

6.4 `type_info` Class

The class `type_info` describes type information generated by the `typeid` operator. The primary functions provided by `type_info` are equality, inequality, `before` and `name`. From `<typeinfo.h>`, the definition is:

```
class type_info {
    public:
        virtual ~type_info( );
        bool operator==( const type_info &rhs ) const;
        bool operator!=( const type_info &rhs ) const;
        bool before( const type_info &rhs ) const;
        const char *name( ) const;
    private:
        type_info( const type_info &rhs );
        type_info &operator=( const type_info &rhs );
};
```

The `before` function compares two types relative to their implementation-dependent collation order. The `name` function returns an implementation-defined, null-terminated, multibyte string, suitable for conversion and display.

The constructor is a private member function, so you cannot create a variable of type `type_info`. The only source of `type_info` objects is in the `typeid` operator.

CHAPTER 7

Cast Operations

This chapter discusses the new cast operators in the C++ standard: `const_cast`, `reinterpret_cast`, `static_cast` and `dynamic_cast`. A cast converts an object or value from one type to another.

7.1 New Cast Operations

The C++ standard defines new cast operations that provide finer control than previous cast operations. The `dynamic_cast<>` operator provides a way to check the actual type of a pointer to a polymorphic class. You can search with a text editor for all new-style casts (search for `_cast`), whereas finding old-style casts required syntactic analysis.

Otherwise, the new casts all perform a subset of the casts allowed by the classic cast notation. For example, `const_cast<int*>(v)` could be written `(int*)v`. The new casts simply categorize the variety of operations available to express your intent more clearly and allow the compiler to provide better checking.

The cast operators are always enabled. They cannot be disabled.

7.2 const_cast

The expression const_cast<*T*>(*v*) can be used to change the const or volatile qualifiers of pointers or references. (Among new-style casts, only const_cast<> can remove const qualifiers.) T must be a pointer, reference, or pointer-to-member type.

```
class A
{
public:
  virtual void f();
  int i;
};
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
  const A a1;
  const_cast<A&>(a1).f( );                // remove const
ip = const_cast<int*> (cvip);    // remove const and volatile
}
```

7.3 reinterpret_cast

The expression reinterpret_cast<*T*>(*v*)changes the interpretation of the value of the expression *v*. It can be used to convert between pointer and integer types, between unrelated pointer types, between pointer-to-member types, and between pointer-to-function types.

Usage of the reinterpret_cast operator can have undefined or implementation-dependent results. The following points describe the only ensured behavior:

- A pointer to a data object or to a function (but not a pointer to member) can be converted to any integer type large enough to contain it. (Type long is always large enough to contain a pointer value on the architectures supported by Sun WorkShop C++.) When converted back to its original type, the value will be the same as it originally was.
- A pointer to a (nonmember) function can be converted to a pointer to a different (nonmember) function type. If converted back to the original type, the value will be the same as it originally was.

- A pointer to an object can be converted to a pointer to a different object type, provided that the new type has alignment requirements no stricter than the original type. If converted back to the original type, the value will be the same as it originally was.
- An lvalue of type *T1* can be converted to a type "reference to *T2*" if an expression of type "pointer to *T1*" can be converted to type "pointer to *T2*" with a reinterpret cast.
- An rvalue of type "pointer to member of X of type *T1*" can be explicitly converted to an rvalue of type "pointer to member of *Y* of type *T2*" if *T1* and *T2* are both function types or both object types.
- In all allowed cases, a null pointer of one type remains a null pointer when converted to a null pointer of a different type.
- The `reinterpret_cast` operator cannot be used to cast away `const`; use `const_cast` for that purpose.
- The `reinterpret_cast` operator should not be used to convert between pointers to different classes that are in the same class hierarchy; use a static or dynamic cast for that purpose. (`reinterpret_cast` does not perform the adjustments that might be needed.) This is illustrated in the following example:

```
class A { int a; public: A(); };
class B : public A { int b, c; };
void use_of_reinterpret_cast( )
{
     A a1;
     long l = reinterpret_cast<long>(&a1);
     A* ap = reinterpret_cast<A*>(l);      // safe
     B* bp = reinterpret_cast<B*>(&a1);    // unsafe
     const A a2;
     ap = reinterpret_cast<A*>(&a2);  // error, const removed
}
```

7.4 static_cast

The expression `static_cast<`*T*`>(`*v*`)` converts the value of the expression *v* to type *T*. It can be used for any type conversion that is allowed implicitly. In addition, any value can be cast to `void`, and any implicit conversion can be reversed if that cast would be legal as an old-style cast.

```
class B           { ... };
class C : public B { ... };
enum E { first=1, second=2, third=3 };
void use_of_static_cast(C* c1 )
{
  B* bp = c1;                  // implicit conversion
  C* c2 = static_cast<C*>(bp); // reverse implicit conversion
  int i = second;              // implicit conversion
  E e = static_cast<E>(i);    // reverse implicit conversion
}
```

The `static_cast` operator cannot be used to cast away `const`. You can use `static_cast` to cast "down" a hierarchy (from a base to a derived pointer or reference), but the conversion is not checked; the result might not be usable. A `static_cast` cannot be used to cast down from a virtual base class.

7.5 Dynamic Casts

A pointer (or reference) to a class can actually point (refer) to any class derived from that class. Occasionally, it may be desirable to obtain a pointer to the fully derived class, or to some other subobject of the complete object. The dynamic cast provides this facility.

Note – When compiling in compatibility mode (`-compat[=4]`), you must compile with `-features=rtti` if your program uses dynamic casts.

The dynamic type cast converts a pointer (or reference) to one class *T1* into a pointer (reference) to another class *T2*. *T1* and *T2* must be part of the same hierarchy, the classes must be accessible (via public derivation), and the conversion must not be

ambiguous. In addition, unless the conversion is from a derived class to one of its base classes, the smallest part of the hierarchy enclosing both *T1* and *T2* must be polymorphic (have at least one virtual function).

In the expression *dynamic_cast*<*T*>(*v*), *v* is the expression to be cast, and *T* is the type to which it should be cast. *T* must be a pointer or reference to a complete class type (one for which a definition is visible), or a pointer to *cv* `void`, where *cv* is an empty string, `const`, `volatile`, or `const volatile`.

7.5.1 Casting Up the Hierarchy

When casting up the hierarchy, if *T* points (or refers) to a base class of the type pointed (referred) to by *v*, the conversion is equivalent to `static_cast<`*T*`>(`*v*`)`.

7.5.2 Casting to `void*`

If *T* is `void*`, the result is a pointer to the complete object. That is, *v* might point to one of the base classes of some complete object. In that case, the result of `dynamic_cast<void*>(`*v*`)` is the same as if you converted *v* down the hierarchy to the type of the complete object (whatever that is) and then to `void*`.

When casting to `void*`, the hierarchy must be polymorphic (have virtual functions). The result is checked at runtime.

7.5.3 Casting Down or Across the Hierarchy

When casting down or across the hierarchy, the hierarchy must be polymorphic (have virtual functions). The result is checked at runtime.

The conversion from *v* to *T* is not always possible when casting down or across a hierarchy. For example, the attempted conversion might be ambiguous, *T* might be inaccessible, or *v* might not point (or refer) to an object of the necessary type. If the runtime check fails and *T* is a pointer type, the value of the cast expression is a null pointer of type *T*. If *T* is a reference type, nothing is returned (there are no null references in C++), and the standard exception `std::bad_cast` is thrown.

For example, this example of public derivation succeeds:

```
class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, public B { };

void simple_dynamic_casts( )
{
  AB  ab;
  B*  bp  = &ab;        // no casts needed
  A*  ap  = &ab;
  AB& abr = dynamic_cast<AB&>(*bp);  // succeeds
  ap = dynamic_cast<A*>(bp);         assert( ap != NULL );
  bp = dynamic_cast<B*>(ap);         assert( bp != NULL );
  ap = dynamic_cast<A*>(&abr);       assert( ap != NULL );
  bp = dynamic_cast<B*>(&abr);       assert( bp != NULL );
}
```

whereas this example fails because base class `B` is inaccessible.

```
class A { public: virtual void f(); };
class B { public: virtual void g(); };
class AB : public virtual A, private B { };

void attempted_casts( )
{
  AB ab;
  B* bp  = (B*)&ab;    // C-style cast needed to break protection
  A* ap  = dynamic_cast<A*>(bp); // fails, B is inaccessible
  assert(ap == NULL);
  AB& abr = dynamic_cast<AB&>(*bp);
  try {
    AB& abr = dynamic_cast<AB&>(*bp); // fails, B is inaccessible
  }
  catch(const bad_cast&) {
    return; // failed reference cast caught here
  }
  assert(0); // should not get here
}
```

In the presence of virtual inheritance and multiple inheritance of a single base class, the actual dynamic cast must be able to identify a unique match. If the match is not unique, the cast fails. For example, given the additional class definitions:

```
class AB_B :     public AB,        public B  { };
class AB_B__AB : public AB_B,      public AB { };
```

Example:

```
void complex_dynamic_casts( )
{
  AB_B__AB ab_b__ab;
  A*ap = &ab_b__ab;
                    // okay: finds unique A statically
  AB*abp = dynamic_cast<AB*>(ap);
                    // fails: ambiguous
  assert( abp == NULL );
                    // STATIC ERROR: AB_B* ab_bp = (AB_B*)ap;
                    // not a dynamic cast
  AB_B*ab_bp = dynamic_cast<AB_B*>(ap);
                    // dynamic one is okay
  assert( ab_bp != NULL );
}
```

The null-pointer error return of `dynamic_cast` is useful as a condition between two bodies of code—one to handle the cast if the type guess is correct, and one if it is not.

```
void using_dynamic_cast( A* ap )
{
  if ( AB *abp = dynamic_cast<AB*>(ap) )
    {            // abp is non-null,
                 // so ap was a pointer to an AB object
                 // go ahead and use abp
      process_AB( abp ); }
  else
    {          // abp is null,
               // so ap was NOT a pointer to an AB object
               // do not use abp
      process_not_AB( ap );
    }
}
```

In compatibility mode (`-compat[=4]`), if runtime type information has not been enabled with the `-features=rtti` compiler option, the compiler converts `dynamic_cast` to `static_cast` and issues a warning. See Section 6.2 “RTTI Options” on page 6-1.

If exceptions have been disabled, the compiler converts `dynamic_cast<T&>` to `static_cast<T&>` and issues a warning. (A dynamic_cast to a reference type requires an exception to be thrown if the conversion is found at run time to be invalid.). For information about exceptions, see Chapter 5.

Dynamic cast is necessarily slower than an appropriate design pattern, such as conversion by virtual functions. See *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma (Addison-Wesley, 1994).

CHAPTER 8

Performance

You can improve the performance of C++ functions by writing those functions in a manner that helps the compiler do a better job of optimizing them. Many books have been written on software performance in general and C++ in particular. For example, see *C++ Programming Style* by Tom Cargill (Addison-Wesley, 1992), *Writing Efficient Programs* by Jon Louis Bentley (Prentice-Hall, 1982), *Efficient C++: Performance Programming Techniques* by Dov Bulka and David Mayhew (Addison-Wesley, 2000), and *Effective C++—50 Ways to Improve Your Programs and Designs*, Second Edition, by Scott Meyers, (Addison-Wesley, 1998). This chapter does not repeat such valuable information, but discusses only those performance techniques that strongly affect the Sun WorkShop C++ compiler.

8.1 Avoiding Temporary Objects

C++ functions often produce implicit temporary objects, each of which must be created and destroyed. For non-trivial classes, the creation and destruction of temporary objects can be expensive in terms of processing time and memory usage. The Sun WorkShop C++ compiler does eliminate some temporary objects, but it cannot eliminate all of them.

Write functions to minimize the number of temporary objects as long as your programs remain comprehensible. Techniques include using explicit variables rather than implicit temporary objects and using reference parameters rather than value parameters. Another technique is to implement and use operations such as `+=` rather than implementing and using only `+` and `=`. For example, the first line below introduces a temporary object for the result of `a + b`, while the second line does not.

```
T x = a + b;
T x( a ); x += b;
```

8.2 Using Inline Functions

Calls to small and quick functions can be smaller and quicker when expanded inline than when called normally. Conversely, calls to large or slow functions can be larger and slower when expanded inline than when branched to. Furthermore, all calls to an inline function must be recompiled whenever the function definition changes. Consequently, the decision to use inline functions requires considerable care.

Do not use inline functions when you anticipate changes to the function definition *and* recompiling all callers is expensive. Otherwise, use inline functions when the code to expand the function inline is smaller than the code to call the function *or* the *application* performs significantly faster with the function inline.

The compiler cannot inline all function calls, so making the most effective use of function inlining may require some source changes. Use the `+w` option to learn when function inlining does not occur. In the following situations, the compiler will *not* inline the function:

- The function contains difficult control constructs, such as loops, switch statements, and try/catch statements. Many times these functions execute the difficult control constructs infrequently. To inline such a function, split the function into two parts, an inner part that contains the difficult control constructs and an outer part that decides whether or not to call the inner part. This technique of separating the infrequent part from the frequent part of a function can improve performance even when the compiler can inline the full function.
- The inline function body is large or complicated. Apparently simple function bodies may be complicated because of calls to other inline functions within the body, or because of implicit constructor and destructor calls (as often occurs in constructors and destructors for derived classes). For such functions, inline expansion rarely provides significant performance improvement, and the function is best left uninlined.
- The arguments to an inline function call are large or complicated. The compiler is particularly sensitive when the object for an inline member function call is itself the result of an inline function call. To inline functions with complicated arguments, simply compute the function arguments into local variables and then pass the variables to the function.

8.3 Using Default Operators

If a class definition does not declare a parameterless constructor, a copy constructor, a copy assignment operator, or a destructor, the compiler will implicitly declare them. These are called default operators. A C-like struct has these default operators. When the compiler builds a default operator, it knows a great deal about the work that needs to be done and can produce very good code. This code is often much faster than user-written code because the compiler can take advantage of assembly-level facilities while the programmer usually cannot. So, when the default operators do what is needed, the program should not declare user-defined versions of these operators.

Default operators are inline functions, so do not use default operators when inline functions are inappropriate (see the previous section). Otherwise, default operators are appropriate when:

- The user-written parameterless constructor would only call parameterless constructors for its base objects and member variables. Primitive types effectively have "do nothing" parameterless constructors.
- The user-written copy constructor would simply copy all base objects and member variables.
- The user-written copy assignment operator would simply copy all base objects and member variables.
- The user-written destructor would be empty.

Some C++ programming texts suggest that class programmers always define all operators so that any reader of the code will know that the class programmer did not forget to consider the semantics of the default operators. Obviously, this advice interferes with the optimization discussed above. The resolution of the conflict is to place a comment in the code stating that the class is using the default operator.

8.4 Using Value Classes

C++ classes, including structures and unions, are passed and returned by value. For Plain-Old-Data (POD) classes, the C++ compiler is required to pass the struct as would the C compiler. Objects of these classes are passed *directly.* For objects of classes with user-defined copy constructors, the compiler is effectively required to construct a copy of the object, pass a pointer to the copy, and destruct the copy after the return. Objects of these classes are passed *indirectly.* For classes that fall between these two requirements, the compiler can choose. However, this choice affects binary compatibility, so the compiler must choose consistently for every class.

For most compilers, passing objects directly can result in faster execution. This execution improvement is particularly noticeable with small value classes, such as complex numbers or probability values. You can sometimes improve program efficiency by designing classes that are more likely to be passed directly than indirectly.

In compatibility mode (`-compat[=4]`), a class is passed indirectly if it has any one of the following:

- A user-defined constructor
- A virtual function
- A virtual base class
- A base that is passed indirectly
- A non-static data member that is passed indirectly

Otherwise, the class is passed directly.

In standard mode (the default mode), a class is passed indirectly if it has any one of the following:

- A user-defined copy constructor
- A user-defined destructor
- A base that is passed indirectly
- A non-static data member that is passed indirectly

Otherwise, the class is passed directly.

8.4.1 Choosing to Pass Classes Directly

To maximize the chance that a class will be passed directly:

- Use default constructors, especially the default copy constructor, where possible.
- Use the default destructor where possible. The default destructor is not virtual, therefore a class with a default destructor should generally not be a base class.
- Avoid virtual functions and virtual bases.

8.4.2 Passing Classes Directly on Various Processors

Classes (and unions) that are passed directly by the C++ compiler are passed exactly as the C compiler would pass a struct (or union). However, C++ structs and unions are passed differently on different architectures.

TABLE 8-1 Passing of Structs and Unions by Architecture

Architecture	Description
SPARC V7/V8	Structs and unions are passed and returned by allocating storage within the caller and passing a pointer to that storage. (That is, all structs and unions are passed by reference.)
SPARC V9	Structs with a size no greater than 16 bytes (32 bytes) are passed (returned) in registers. Unions and all other structs are passed and returned by allocating storage within the caller and passing a pointer to that storage. (That is, small structs are passed in registers; unions and large structs are passed by reference.) As a consequence, small value classes are passed as efficiently as primitive types.
IA platforms	Structs and unions are passed by allocating space on the stack and copying the argument onto the stack. Structs and unions are returned by allocating a temporary object in the caller's frame and passing the address of the temporary object as an implicit first parameter.

8.5 Cache Member Variables

Accessing member variables is a common operation in C++ member functions.

The compiler must often load member variables from memory through the `this` pointer. Because values are being loaded through a pointer, the compiler sometimes cannot determine when a second load must be performed or whether the value loaded before is still valid. In these cases, the compiler must choose the safe, but slow, approach and reload the member variable each time it is accessed.

You can avoid unnecessary memory reloads by explicitly caching the values of member variables in local variables, as follows:

- Declare a local variable and initialize it with the value of the member variable.
- Use the local variable in place of the member variable throughout the function.

- If the local variable changes, assign the final value of the local variable to the member variable. However, this optimization may yield undesired results if the member function calls another member function on that object.

This optimization is most productive when the values can reside in registers, as is the case with primitive types. The optimization may also be productive for memory-based values because the reduced aliasing gives the compiler more opportunity to optimize.

This optimization may be counter-productive if the member variable is often passed by reference, either explicitly or implicitly.

On occasion, the desired semantics of a class requires explicit caching of member variables, for instance when there is a potential alias between the current object and one of the member function's arguments. For example:

```
complex& operator*= (complex& left, complex& right)
{
  left.real = left.real * right.real + left.imag * right.imag;
  left.imag = left.real * right.imag + left.image * right.real;
}
```

will yield unintended results when called with:

```
x*=x;
```

CHAPTER 9

Multithreaded Programs

This chapter explains how to build multithreaded programs. It also discusses the use of exceptions and explains how to share C++ Standard Library objects across threads.

For more information about multithreading, see the *Multithreaded Programming Guide*, the *C++ Library Reference*, the *Tools.h++ User's Guide*, and the *Standard C++ Library User's Guide*.

9.1 Building Multithreaded Programs

All libraries shipped with the C++ compiler are multithreading-safe. If you want to build a multithreaded application, or if you want to link your application to a multithreaded library, you must compile and link your program with the `-mt` option. This option passes `-D_REENTRANT` to the preprocessor and passes `-lthread` in the correct order to `ld`. For compatibility mode (`-compat[=4]`), the `-mt` option ensures that `libthread` is linked before `libC`. For standard mode (the default mode), the `-mt` option ensures that `libthread` is linked before `libCrun`.

Do not link your application directly with `-lthread` because this causes `libthread` to be linked in an incorrect order.

The following example shows the correct way to build a multithreaded application when the compilation and linking are done in separate steps:

```
example% CC -c -mt myprog.cc
example% CC -mt myprog.o
```

The following example shows the wrong way to build a multithreaded application:

```
example% CC -c -mt myprog.o
example% CC myprog.o -lthread <- libthread is linked incorrectly
```

9.1.1 Indicating Multithreaded Compilation

You can check whether an application is linked to `libthread` or not by using the `ldd` command:

```
example% CC -mt myprog.cc
example% ldd a.out
libm.so.1 =>      /usr/lib/libm.so.1
libCrun.so.1 =>   /usr/lib/libCrun.so.1
libw.so.1 =>      /usr/lib/libw.so.1
libthread.so.1 => /usr/lib/libthread.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libdl.so.1 =>     /usr/lib/libdl.so.1
```

9.1.2 Using C++ Support Libraries With Threads and Signals

The C++ support libraries, `libCrun`, `libiostream`, `libCstd`, and `libC` are multithread safe but are not `async` safe. This means that in a multithreaded application, functions available in the support libraries should not be used in signal handlers. Doing so can result in a deadlock situation.

It is not safe to use the following in a signal handler in a multithreaded application:

- Iostreams
- `new` and `delete` expressions
- Exceptions

9.2 Using Exceptions in a Multithreaded Program

The current exception-handling implementation is safe for multithreading; exceptions in one thread do not interfere with exceptions in other threads. However, you cannot use exceptions to communicate across threads; an exception thrown from one thread cannot be caught in another.

Each thread can set its own `terminate()` or `unexpected()` function. Calling `set_terminate()` or `set_unexpected()` in one thread affects only the exceptions in that thread. The default function for `terminate()` is `abort()` for the main thread, and `thr_exit()` for other threads (see Section 5.5 “Specifying Runtime Errors” on page 5-7).

Note – Thread cancellation (`pthread_cancel(3T)`) results in the destruction of automatic (local nonstatic) objects on the stack. When a thread is cancelled, the execution of local destructors is interleaved with the execution of cleanup routines that the user has registered with `pthread_cleanup_push()`. The local objects for functions called after a particular cleanup routine is registered are destroyed before that routine is executed.

9.3 Sharing C++ Standard Library Objects Between Threads

The C++ Standard Library (`libCstd`), which is multithread-safe, makes sure that the internals of the library work properly in a multithreading environment. You will still need to lock around any library objects that you yourself share between threads (except for `iostreams` and `locale` objects).

For example, if you instantiate a string, then create a new thread and pass that string to the thread by reference, then you must lock around write access to that string, since you are explicitly sharing the one string object between threads. (The facilities provided by the library to accomplish this task are described below.)

On the other hand, if you pass the string to the new thread by value, you do not need to worry about locking, even though the strings in the two different threads may be sharing a representation through Rogue Wave’s “copy on write” technology.

The library handles that locking automatically. You are only required to lock when making an object available to multiple threads explicitly, either by passing references between threads or by using global or static objects.

The following describes the locking (synchronization) mechanism used internally in the C++ Standard Library to ensure correct behavior in the presence of multiple threads.

The interface to this facility (including the names of files, macros, classes, and any class members) as well as the implementation are an internal detail of the library and are subject to change without notice. Backward compatibility is not guaranteed.

Two synchronization classes provide mechanisms for achieving multithreaded safety; `_RWSTDMutex` and `_RWSTDGuard`.

The `_RWSTDMutex` class provides a platform-independent locking mechanism through the following member functions:

- `void acquire()`—Acquires a lock on self, or blocks until such a lock can be obtained.
- `void release()`—Releases a lock on self.

```
class _RWSTDMutex
{
public:
    _RWSTDMutex ();
    ~_RWSTDMutex ();
    void acquire ();
    void release ();
};
```

The `_RWSTDGuard` class is a convenience wrapper class that encapsulates an object of `_RWSTDMutex` class. An `_RWSTDGuard` object attempts to acquire the encapsulated mutex in its constructor (throwing an exception of type `::thread_error`, derived from `std::exception on error`), and releases the mutex in its destructor (the destructor never throws an exception).

```
class _RWSTDGuard
{
public:
    _RWSTDGuard (_RWSTDMutex&);
    ~_RWSTDGuard ();
};
```

Additionally, you can use the macro `_RWSTD_MT_GUARD(mutex)` (formerly `_STDGUARD`) to conditionally create an object of the `_RWSTDGuard` class in multithread builds. The object guards the remainder of the code block in which it is defined from being executed by multiple threads simultaneously. In single-threaded builds the macro expands into an empty expression.

The following example illustrates the use of these mechanisms.

```
#include <rw/stdmutex.h>;

//
// An integer shared among multiple threads.
//
int I;

//
// A mutex used to synchronize updates to I.
//
_RWSTDMutex I_mutex;

//
// Increment I by one.  Uses an _RWSTDMutex directly.
//

void increment_I ()
{
   I_mutex.acquire();  // Lock the mutex.
   I++;
   I_mutex.release();  // Unlock the mutex.
}

//
// Decrement I by one.  Uses an _RWSTDGuard.
//

void decrement_I ()
{
   _RWSTDGuard guard(I_mutex);  // Acquire the lock on I_mutex.
   --I;
   //
   // The lock on I is released when destructor is called on guard.
   //
}
```

Index

F

H

I

M

O

P

R

S

T

U

V