C++ Programming Guide

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

Part No: 805-4955
Revision A, February 1999

Contents

Preface

This manual tells you how to use C++ 5.0 features to write more efficient programs.

Who Should Use This Book

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

How This Book Is Organized

This book contains the following chapters:

Chapter 1, "Introduction," 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 Sun C++ 5.0 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 9, "Performance," explains how to improve the performance of C++ functions.

Multiplatform Release

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

The Sun™ WorkShop™ documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments on:

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

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

C++ Compiler Related Books

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

- *C++ User's Guide* instructs you in the use of the C++ 5.0 compiler and provides detailed information on command-line options.
- *C++ Library Reference* describes the C++ libraries, including the C++ Standard Library, the `Tools.h++` Class Library, the Sun WorkShop Memory Monitor, and the iostream and complex libraries.
- *C++ Migration Guide* explains what you need to know when moving from 4.0, 4.0.1, 4.1, or 4.2 versions of the C++ compiler to the C++ 5.0 version.
- *Tools.h++ User's Guide* discusses use of the C++ classes for enhancing the efficiency of your programs.
- *Tools.h++ Class Library Reference* provides details on the `Tools.h++` class library.

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

Other Sun WorkShop Books

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

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

Solaris Books

The following Solaris manuals and guides provide additional useful information:

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

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

Commercially Available Books

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

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

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

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

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

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

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

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

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

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

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

Ordering Sun Documents

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

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

Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents
- Online help and release notes

Using the `docs.sun.com` Web site

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

Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

Note - To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install*.

Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*

- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

1. **Open the following file through your HTML browser:**

   install-directory`/SUNWspro/DOC5.0/lib/locale/C/html/index.html`

   Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

   The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. **Open a document in the index by clicking the document's title.**

Accessing Sun WorkShop Online Help and Release Notes

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

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

Man Pages

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

```
demo% man topic
```

Man pages are in:

opt-install-dir`/SUNWspro/man`

Table P–1 lists and describes the C++ man pages.

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

TABLE P–1 C++ Man Pages

Title	Description
CC	Drives the C++ compilation system
cartpol	Provides Cartesian/polar functions in the C++ complex number math library
cplx.intro	Introduces the C++ complex number math library
cplxerr	Provides complex error-handling functions in the C++ complex number math library
cplxops	Provides arithmetic operator functions in the C++ complex number math library
cplextrig	Provides trigonometric operator functions in the C++ complex number math library
demangle	Decodes a C++ encoded symbol name
filebuf	Buffer class for file I/O
fstream	Provides stream class for file I/O
istream	Supports formatted and unformatted input
ios	Provides basic iostream formatting
ios.intro	Introduces iostream man pages
manip	Provides iostream manipulators
ostream	Supports formatted and unformatted output
queue	Provides list management for task library
sbufprot	Provides protected interface of streambuffer base class
sbufpub	Provides public interface of streambuffer base class

TABLE P–1 C++ Man Pages *(continued)*

Title	Description
sigfpe	Allows signal handling for specific SIGFPE codes
ssbuf	Provides buffer class for character arrays
stdarg	Handles variable argument list
stdiobuf	Provides buffer and stream classes for use with C stdio
stream_locker	Provides class used for application level locking of iostream class object
stream_MT	Base class that provides dynamic changing of iostream class object to and from MT safely
strstream	Provides stream class for I/O using character arrays
varargs	Handles variable argument list
vector	Provides generic vector and stack

Table P–2 lists man pages that contain information related to the C++ compiler.

TABLE P–2 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

TABLE P–2 Man Pages Related to C++ *(continued)*

Title	Description
lex	Generates lexical analysis programs
rpcgen	Generates C/C++ code to implement an RPC protocol
version	Displays version identification of object file or binary
yacc	Converts a context-free grammar into a set of tables for a simple automaton that executes an LALR(1) parsing algorithm

README file

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

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

README files are in:

opt-install-dir`/SUNWspro/READMEs`

To view the C++ compiler `README` file, type:

```
%CC -readme
```

What Typographic Changes Mean

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

TABLE P–3 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output.	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output.	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value.	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Compiler options and commands use the following conventions:		
`[ ]`	Square brackets contain arguments that are optional.	`-xO[n]`
( )	Parentheses contain a set of choices for a required option.	`-d(y|n)`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be used at one time.	–d(y | n)
`...`	The ellipsis indicates omission in a series.	`-xinline=`*f1*[,...*fn*]
`%`	The percent sign indicates the word has a special meaning.	`-ftrap=%all, no%division`
<>	In ASCII files, such as the `README` file, angle brackets contain a variable that must be replaced by an appropriate value.	`-xtemp=<`*dir*`>`

Shell Prompts in Command Examples

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

TABLE P–4 System Prompts

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

CHAPTER 1

Introduction

The Sun C++ compiler, `CC`, described in this book (and the companion book, *C++ User's Guide*) is available under the Solaris 2.5.1, 2.6, and Solaris 7 operating environments on the hardware platforms in SPARC and x86. Sun C++ 5.0 implements the language and libraries described in the C++ International Standard.

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.

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.

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.

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.

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.

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 alternate 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.

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.

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
```

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
```

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
```

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.

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.

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.

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.

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.

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 "Idempotent Header Files" on page 9.)

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 "Idempotent Header Files" on page 9.)

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 the 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.*

Pragma Forms

Note - Pragmas are not part of any C++ standard.

The various forms of a `CC` 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 recognized by `CC` 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.

Pragma Reference

`#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;
```

`#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.

`#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.

`#pragma ident` *string*

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

`#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.

TABLE 3–1 Strictest Alignment by Platform

Platform	Strictest Alignment
x86	4
SPARC generic, v7, v8, v8a, v8plus, v8plusa	8
SPARC v9, v9a	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.

`#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.

`#pragma weak` *function-name1* `[=` *function-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 following directive defines `bar` to be a weak symbol. No error messages are generated if the linker cannot find a definition for a function named `bar.`

```
#pragma weak bar
```

The following directive instructs the linker to resolves any references to `bar` to `bar` if it is defined anywhere in the program, and to `foo` otherwise.

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

You must declare a function before you use it in a `weak` pragma. For example:

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

The effects of using `#pragma weak` are:

- If your program calls but does not define *function-name1*, the linker uses the definition from the library.
- If your program defines its own version of *function-name1*, then the program definition is used, and the weak global definition of *function-name1* in the library is not used.
- If the program directly calls *function-name2*, the definition from the library is used; a duplicate definition of *function-name2* causes an error.

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

Note - The names in the pragma must be the names a seen by the linker, which means the “mangled” name if the function has C++ linkage.

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 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.

Function Templates

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

C++ 5.0 does not support non-type template parameters for function templates.

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 template declarations and definitions stands for some to-be-determined type.

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*.

C++ 5.0 does not support non-type template parameters for function templates. For example, the following template is not supported because its argument is an expression instead of a type.

```
template <int count> void foo( )  // unsupported non-type parameter
{
    int x[count]
    for (int i = 0; i < count; ++i )
        // ... do something with x
}

foo<10>(); // call foo with template argument 10; unsupported
```

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 infers 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 ); }
```

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.

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 integer` as an argument.

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

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

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.

For Function Members

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 ) overflow++; }
```

For Static Data Members

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;
```

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 "Template Instantiation" on page 23).

```
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( );
```

.

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.

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.

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.

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.

For 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 );
```

For 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.

For 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( );
```

.

For 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;
```

.

Template Composition

You can use (but not define) 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
```

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.

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.

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 );
```

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; } }
```

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.

Template Problem Areas

This section describes problems you might encounter when using templates.

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 mean the same thing 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.

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:

```
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.

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.

```
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 it ahead of 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
```

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
```

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:

```
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 currently implemented in the Sun C++ compiler, and 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.

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.

Using Exception Handling Keywords

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

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

`try`

A `try` block is a group of C++ statements, normally enclosed in braces `{ }`, which 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.

`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:

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

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
 }
}
```

Implementing Exception Handlers

To implement an exception handler, do 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, this is what takes place:

- 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.
- If `try` blocks are nested, and there is no match, control passes from the innermost `catch` block to the nearest `catch` block surrounding the `try` block.
- If no matching `catch` block is found in the current function, any *automatic* (local nonstatic) objects in the current function are destroyed and the function exits immediately. A search for a matching `catch` block continues with the function that called the current function. This process continues up to function `main`.
- 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.

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.

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.

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` issued in this context returns from the function containing the `catch` block.

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.

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 “Calling the `uncaught_exception()` Function” on page 41.

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 via any exception that is not allowed by an exception specification, it results in a call to the predefined function `unexpected()`. By default, `unexpected()` calls `abort()` to exit the program. You can change this default behavior by calling the `set_unexpected()` function. See "`set_unexpected()` " on page 40.

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.

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.

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()`.

`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 via 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 - Selecting a `terminate()` replacement that returns to its caller, or that does not terminate the program or thread, is an error.

---

set_unexpected()

You can modify the default behavior of `unexpected()` by calling the function `set_unexpected()`:

```
// 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 via 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 - Selecting an `unexpected()` replacement that returns to its caller is an error.

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 program termination due to a destructor throwing an exception while another exception is still active is to design your program so that destructors do not need to throw exceptions.)

Matching Exceptions With Handlers

A handler type `T` matches a `throw` type `E` if any 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.

Checking Access Control in Exceptions

The compiler performs the following check on access control on 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 "Matching Exceptions With Handlers" on page 41.

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}
```

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.

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 header provides the following 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 (option `-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.

Building Shared Libraries With Exceptions

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

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

Using Exceptions in a Multithreaded Environment

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 “Specifying Runtime Errors” on page 38).

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.

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.

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.

RTTI Options

For C++ 5.0 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).

For C++ 5.0 in standard mode (the default mode) RTTI does not have a significant impact on program compilation or execution. In standard mode, RTTI is always enabled.

`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/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 raises a `bad_typeid` exception when given a null pointer.

`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 recently approved C++ standard: `const_cast`, `reinterpret_cast`, `static_cast` and `dynamic_cast`.

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 in C++ 5.0. They cannot be disabled.

`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 int A::* cimp;
extern const volatile int* cvip;
extern int* ip;
void use_of_const_cast( )
{
  const A a1;
  const_cast<A&>(a1).f( );                // remove const
  a1.*(const_cast<int A::*> cimp) = 1;    // remove const
  ip = const_cast<int*> cvip;    // remove const and volatile
}
```

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 big enough to contain a pointer value on the architectures supported by Sun 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 when converted to a null pointer of a different type remains a null pointer.
- 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; };
   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
   }
```

static_cast

The expression `static_cast<`*T*`>(`*v*`)` converts the value of the expression *v* to that of type *T*. It can be used for any type conversion that is allowed implicitly. In addition, any value may 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.

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.

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`.

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*`)`.

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.

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.

When you assume the following declarations:

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

The following function succeeds:

```
void simple_dynamic_casts( )
{
  AB  ab;
  B*  bp  = (B*)&ab;    // cast needed to break protection
  A*  ap  = &ab;        // public derivation, no cast needed
  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 );
}
```

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 { };
```

The following function succeeds:

```
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 Chapter 5.

If exceptions have been disabled, the compiler converts `dynamic_cast<T&>` to `static_cast<T&>` and issues a warning. The dynamic cast to a reference might require an exception in normal circumstances. See Chapter 4.

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 Tom Cargill, *C++ Programming Style*, Addison-Wesley, 1992, Jon Louis Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982, and Scott Meyers, *Effective C++*, Addison-Wesley, 1992.) This chapter does not repeat such valuable information, but discusses only those performance techniques that strongly affect the Sun C++ compiler.

Avoiding Temporary Objects

C++ functions often produce many implicit temporary objects, each of which must be created and destroyed. For non-trivial classes, this creation and destruction can get expensive. The Sun 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;
```

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.

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). Default operators cannot be virtual. 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.

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 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 (`-compat=5`), a class is passed indirectly if it has any 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.

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.

Passing Classes Directly on Various Processors

Classes (and unions) 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.

On 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.)

On 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.

On x86, 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.

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;
```

Index