C++ User's Guide

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.:802-5660-10
Revision A, December 1996

Contents

Tables

Preface

This manual, *C++ User's Guide*, explains how to use the C++ compiler and some of the newer C++ language features. This manual complements the C++ documentation set described in the "C++ Documentation" section, below.

Audience

The audience for this book includes software developers writing programs in the C++ language who are familiar with compiled program development and C++ language basics. This book is not a basic text in C++ programming.

Organization of This Book

This book contains the following chapters:

- Chapter 1, "Introducing the C++ Compiler," gives an overview of the compiler.
- Chapter 2, "Using the C++ Compiler," describes the options available with the compiler.
- Chapter 3, "Templates," describes the templates available with the compiler and how to use them.
- Chapter 4, "Exception Handling," explains exception handling as currently implemented in the compiler.

- Chapter 5, “Runtime Type Information,” explains the RTTI options supported by the compiler.
- Chapter 6, ”Cast Operations,” describes new cast operations.
- Chapter 7, “Moving from C to C++,” describes how to move programs from C to C++.
- Chapter 8, “Fortran 77 Interface,” describes the Fortran interface with C++.

The book also contains these appendixes:

- Appendix A, “Migration Guide,” describes how to change old C++ code to new.
- Appendix B, ”Code Samples,” gives examples of C++ programs.
- Appendix C, “Localization Support,” describes support for languages other than English.

C++ Documentation

Although there is no prerequisite reading for this guide, you should have access to C++ reference books such as *The C++ Programming Language* by Bjarne Stroustrup. You should also have access to the documents described in the following sections.

.

Table P-1 Summary of C++ Compiler Documentation and Its Location

Document	**On-line Books**	**HTML**	**On-line ASCII**	**On-line PostScript**	**Hard copy**
C++ User’s Guide	X	X			X
C++ Quick Reference					X
C++ Library Reference	X	X			X
Tools.h++ User’s Guide	X	X			X
Tools.h++ Class Library Reference	X	X			X
man pages			`/opt/SUNWspro /man`		

Table P-1 Summary of C++ Compiler Documentation and Its Location *(Continued)*

Document	**On-line Books**	**HTML**	**On-line ASCII**	**On-line PostScript**	**Hard copy**
C++ `README` File			`/opt/SUNWspro /READMEs/C++`		
Sun WorkShop Installation and Licensing Guide	X	X			X
Sun WorkShop Quick Install for Solaris		X			X
"What Every Computer Scientist Should Know About Floating-Point Arithmetic" white paper				`/opt/SUNWspro /READMEs /floating-point.ps`	

Manuals

In addition to the *C++ User's Guide*, the following books are included in the C++ documentation set.

- *C++ Library Reference.*This guide gives information about how to use the complex, coroutine, and iostream libraries.
- *Sun WorkShop Installation and Licensing Guide.*This manual tells you how to install Sun™ C++ software, in addition to other software, on the Solaris operating environment.
- *Tools.h++ User's Guide*. This manual introduces you to and tells you how to use the `Tools.h++` class library.
- *Tools.h++ Class Library Reference*. The `Tools.h++` class library is a set of C++ classes that can greatly simplify your programming while maintaining the efficiency for which C is famous.

Commercially Available Books

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

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)

A C++ Primer, Second Edition, Stanley B. Lippman (Addison-Wesley, 1989)

Effective C++—50 Ways to Improve Your Programs and Designs, Scott Meyers (Addison-Wesley, 1996)

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

The C++ Programming Language, Second Edition, Bjarne Stroustrup (Addison-Wesley, 1991)

Online Documentation

Online Books

Certain manuals are provided in online viewing tools that take advantage of dynamically linked headings and cross-references. Online documentation enables you to electronically jump from one subject to another and to search for topics by using a word or phrase.

Online documentation for this product is installed separately. See *Sun WorkShop Installation and Licensing Guide* for further information on installation.

- *Programming Utilities and Libraries*. This manual covers a few of the tools that can aid you in programming. These include:

  `lex` — Generates programs used in simple lexical analysis of text, solves problems by recognizing different strings of characters.

  `yacc` — Takes a description of a grammar and generates a C function to parse the input stream according to that grammar.

  `prof` — Produces an execution profile of the modules in a program.

  `make`—Automatically maintains, updates, and regenerates related programs and files.

  System V `make`—Describes a version of `make` that is compatible with older versions of the tool.

`sccs`—Allows you to control access to shared files and to keep a history of changes made to a project.

`m4`—Macro language processor.

This manual is bundled with the operating system documentation.

- *Solaris Linker and Libraries Manual*. This book gives information on linking libraries.

Man Pages

Each man page concisely explains a single subject, which could be a user command or library function. Man pages are in:

opt install dir / `SUNWspro` / `man`

Table P-2 lists and describes the C++ man pages.

Note – Before you use the `man` command, insert the name of the directory in which you have chosen to install the C++ compiler at the beginning of your search path. Doing 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-2 C++ Man Pages

Title	**Description**
CC	Displays 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

Table P-2 C++ Man Pages *(Continued)*

Title	**Description**
demangle	Decodes a C++ encoded symbol name
filebuf	Buffer class for file I/O
fstream	Provides stream class for file I/O
generic	Provides generic macro definitions used for creating generic types
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
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
task	Provides coroutines in the C++ task library
task.intro	Introduces coroutine library man pages
tasksim	Provides histogram and random numbers for the task library
varargs	Handles variable argument list
vector	Provides generic vector and stack

Table P-3 lists man pages that contain information related to the C++ compiler.

Table P-3 C++-Related Man Pages

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

`README` *file*

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

- New/changed features
- Software incompatibilities
- Current software bugs
- Documentation errata

View the `README` file by typing
`CC -readme`.

Other `README`s:

Related information can be found in the `READMEs` directory, which is located at `/opt/SUNWspro/READMEs` in a standard installation.

C++ Migration Guide

The *C++ Migration Guide* helps you migrate your code from `cfront`-based C++ 3.0 to the current compiler. This manual is displayed when you type `CC -migration`. (The *C++ Migration Guide* is also found in Appendix A of this manual.)

Articles

"What Every Computer Scientist Should Know About Floating-Point Arithmetic"

A floating-point white paper by David Goldberg. This paper can be found in the `README` directory:

opt-install-dir`/SUNWspro/READMEs`

Conventions in Text

The following table describes the typographic conventions and symbols used in this book.

Table P-4 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. `system% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`system%` **`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.

Code samples are included in boxes and may display the following:

`%`	C shell prompt	`system%`

Table P-4 Typographic Conventions *(Continued)*

Typeface or Symbol	Meaning	Example
`$`	Bourne and Korn shell prompt	`system$`
`#`	Superuser prompt, all shells	`system#`
`[ ]`	Square brackets contain arguments that can be optional or required.	-d[y | n]
`|`	The "pipe" or "bar" symbol separates arguments, only *one* of which may be used at one time.	-d[y | n]
`,`	The comma separates arguments, *one or more* of which may be used at one time.	-xinline=[*f1*,...*fn*]
`:`	The colon, like the comma, is sometimes used to separate arguments.	-R*dir*[:*dir*]
`...`	The ellipsis indicates omission in a series.	-xinline=[*f1*,...*fn*]
`%`	The percent sign indicates the word following it has a special meaning.	-ftrap=%all
`<>`	In ASCII files, such as the `README` file, angle brackets contain a variable that must be replaced by an appropriate value.	-xtemp=<dir>

The C++ Compiler 1

This chapter provides a brief conceptual overview of C++ and the C++ compiler, with particular emphasis on the areas of difference and similarity with C. Chapter 7, “Moving From C to C++,” summarizes issues important to C programmers moving to C++.

Operating Environments

For an explanation of the specific operating environments supported in this release, refer to the `README` file. You can view this file by typing the command

```
% CC -readme
```

Standards Conformance

The compiler implements the C++ language as described in the *C++ Annotated Reference Manual* (ARM). It also conforms to selected extensions of the January 1996 *C++ Draft Working Paper* (ISO).

Organization of the Compiler

The C++ compiler package consists of a front end, optimizer, code generator, assembler, template pre-linker, and link editor. The CC command invokes each of these components automatically unless you use command-line options to specify otherwise. Figure 1-1 shows the C++ compilation system

Figure 1-1 Organization of the C++ Compilation System.

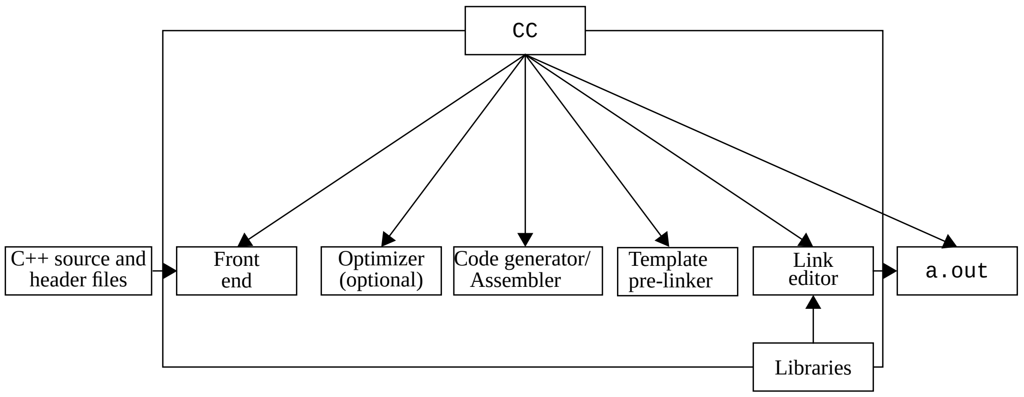

Table 1-1 summarizes the components of the C++ compilation system.

Table 1-1 Components of the C++ Compilation System

Component	Description	Notes on Use
`ccfe`	Front end (Compiler preprocessor and compiler)	
`iropt`	Code Optimizer	*(SPARC)* `-O`, `-xO[2-5]`, `-fast`
`cg386`	Intermediate language translator	*(x86)* Always invoked
`cgppc`	Intermediate language translator	*(PowerPC)* Always invoked
`inline`	Inline expansion of assembly language templates	`.il` file specified
`mwinline`	Automatic inline expansion of functions	*(x86) (PowerPC)* `-xO4`, `-xinline`
`fbe`	Assembler	
`cg`	Code generator, inliner, assembler	*(SPARC)*

Table 1-1 Components of the C++ Compilation System *(Continued)*

Component	**Description**	**Notes on Use**
`codegen`	Code generator	*(x86) (PowerPC)*
`tdb_link`	Template pre-linker	
`ld`	Non-incremental editor	
`ild`	Incremental link editor	`-g`, `-xildon`

The C++ compiler package also includes:

- On-line `README` files containing the latest known software and documentation bugs and other late-breaking information
- Man pages—single-page (generally), online documentation that concisely describes a user command or library function
- The C++ name demangling tool set (`dem` and `c++filt`)
- C++ library functions for stream I/O, complex arithmetic, tasking, and other operations
- `Tools.h++` class library—a set of C++ classes that can simplify your programming
- `#include` files—files you can use for standard features such as signals and ctype with C++ programs

C++ Tools

Most of the C++ tools are now incorporated into traditional UNIX® tools. These tools are bundled with the operating system. They are:

- `lex`—Generates programs used in simple lexical analysis of text
- `yacc`—Generates a C function to parse input stream according to syntax
- `prof`—Produces an execution profile of modules in a program
- `gprof`—Profiles by procedure

Please see *Profiling Tools* and associated man pages for further information on these UNIX tools.

1

The C++ Language

This version of C++ supports the C++ language as described in *The C++ Programming Language* by Margaret Ellis and Bjarne Stroustrup, with a few deletions and a number of extensions.

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

- Stronger type checking
- Extensive data abstraction features
- Support for object-oriented programming

This last feature, particularly, allows good design of modular and extensible interfaces among program modules.

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. Pascal always prohibits attempts to use data of the wrong type; the C++ compiler produces errors in some cases, but in others converts data to the correct type.

Rather than having the C++ compiler do these automatic conversions, you can explicitly convert between types, just as you can in C.

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. This action may lead to ambiguous situations. If the resolution is not clear at compile time, the compiler issues an “ambiguity” error.

Classes and Data Abstraction

A class is a user-defined type. If you are a C programmer, think of a class as an extension of the idea of `struct` in C. Like the predefined types in C, classes are defined not only with data storage but also with operations that apply to the data. In C++, these operations include operators and functions. For example, if you define a class `carrot`, you can define the `+` operator so it has a meaning when used with `carrots`. If `carrot1` and `carrot2` are objects of the type `carrot`, then the expression:

```
carrot1 + carrot2
```

has a value determined by your definition of `+` in the case of `carrots`. This definition does not override the original definition of `+`; as with overloaded function names, the compiler determines from context what definition of `+` it should use. Operators with extra definitions like this are called *overloaded operators*.

In addition to operators, classes may have member functions, functions that exist to operate on objects of that class.

C++ provides classes as a means for *data abstraction*. You decide what types (classes) you want for your program's data and then decide what operations each type needs.

The members of a class can be divided into three parts: `public`, `private`, and `protected`. The `public` part is available to any function; the `private` part is available only to member and friend functions; the `protected` part is available to members, friends, and members of derived classes.

Object-Oriented Features

A program is object-oriented when it is designed with classes, and the classes are organized so that common features are embodied in base classes, sometimes called *parent classes*. The feature that makes this possible is inheritance. A class in C++ can inherit features from one base class or from several. A class that has a base class is said to be derived from the base class.

Native Language Support

This release of C++ supports the development of applications in languages other than English, including most European languages and Japanese. As a result, you can easily switch your application from one native language to another. This feature is known as *internationalization*.

In general, the C++ compiler implements internationalization as follows:

- C++ recognizes ASCII characters from international keyboards (in other words, it has keyboard independence and is 8-bit clean).
- C++ allows the printing of some messages in the native language.
- C++ allows native language characters in comments, strings, and data.

Variable names cannot be internationalized and must be in the English character set.

You can change your application from one native language to another by setting the locale. For information on this and other native language support features, see the operating system documentation.

Compatibility With C

C++ is highly compatible with C. The language was purposely designed this way; 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 from C; an ordinary C program may not be accepted by the C++ compiler without some modifications. Chapter 7, "Moving From C to C++," discusses what you must know to move from programming in C to programming in C++.

Even though the differences between C and C++ are most evident in the way you can design interfaces between program modules, 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++:

- Defined constants allow you to avoid the preprocessor and use named constants in your program.
- Function prototypes are required.
- Free store operators `new` and `delete` create dynamic variables.
- References, alternate "handles" on the same object, are automatically dereferenced pointers and act like an alternate name for a variable. You can use references as function parameters.

- Functional syntax for type coercions is supported.
- Programmer-defined automatic type conversion is allowed.
- Variable declarations are allowed anywhere, not just at the beginning of the block.
- A new comment delimiter begins a comment that continues to the end of the line.
- The name of an enumeration or class is also automatically a type name.
- Default values can be assigned to function parameters.
- Inline functions can ask the compiler to replace a function call with the function body, improving program efficiency.

Using the C++ Compiler 2

This chapter demonstrates how to compile and link your code, as well as how to use compiler pragmas and options.

Compiler Commands

Before using the `CC` command, insert the name of the directory in which you have chosen to install the C++ compiler at the beginning of your search path. The default path is:

```
/opt/SUNWspro/bin
```

To compile a simple program, `myprog`, enter the following command:

```
% CC myprog.cc -o myprog
```

The resulting executable file is called `myprog` because this command line uses the `-o` *name* argument. Without that argument, the executable file has the default name, `a.out`.

The possible file name extensions for the source file are: `.c`, `.C`, `.cc`, `.cpp`, or `.cxx`.

Command Syntax

The general syntax of the compiler command line is:

`CC` `[`*options*`]` *list_of_files* `[`**`-l`***x*`]`

Items in square brackets indicate optional parameters. The brackets are not part of the command. The *options* are a list of option keywords prefixed by dash (–). Some keyword options take the next item in the list as an argument. The *list_of_files* is a list of source, object, or library file names separated by blanks.

- `–l`*x* is the option to link with library `lib`*x*`.a`. It is always safer to put `–l`*x* after the list of file names to insure the correct order libraries are searched.
- In general, processing of the compiler options is from left to right, allowing selective overriding of macro options (options that include other options).
  - The above rule does not apply to linker options.
  - The `–I`, `–L`, and `–R` options accumulate, not override.

Source files, object files, and libraries are compiled and linked in the order in which they appear on the command line.

Compiling and Linking

The sample program `testr`, used in Appendix A, consists of two modules: the main program module, `testr.cc`, and the string class module, `str.cc` and `str.h`.

When you have a second module like the string class module, both the implementation part of the second module and the main program module must include the header file for the second module. For example, `testr.cc` and `str.cc` include the header file, `str.h`, with a line like:

```
#include "str.h"
```

If there is no object file for the second module, you can compile the second module and link it with the program with a command line like:

```
% CC testr.cc str.cc -o testr
```

Alternately, you can create an object file for the second module with this command line:

```
% CC -c str.cc
```

This line does not invoke the linker and results in an object file called `str.o`.

When there is an object file for the unit, you can compile the program and link it with the unit, as follows:

```
% CC str.o testr.cc -o testr
```

Compatibility Between C++ 4.0.1 and C++ 4.1

The object files produced by C++ 4.1 and C++ 4.0.1 are binary compatible for most cases. That is, object files created by C++ 4.0.1 can be linked using the C++ 4.1 compiler, and object files created by C++ 4.1 can be linked using the C++ 4.0.1 compiler without any problems. However, this may not be true if you are using exceptions and link statically with `libC`. Note the following cases:

- If you are not using exceptions, object files produced by the two compilers are binary compatible. You can create binary files with either compiler, and link using either compiler, without any compatibility problems. This holds true whether you link statically or dynamically with `libC.`
- If you are using exceptions, the result will depend on whether you are linking statically or dynamically with `libC`.
  - If you link statically with `libC`, the object files produced by C++ 4.1 are not binary compatible with the `libC.a` of C++ 4.0.1. That is, if you create binary files with C++ 4.1, and link them using the C++ 4.0.1 compiler, the resulting executable may not work. You may get a core dump. However, binary files created with the C++ 4.0.1 compiler will work with the `libC.a` of C++ 4.1. If you create binary files with the C++ 4.0.1 compiler, and link them using the C++ 4.1 compiler, there will be no compatibility problems.
  - If you link dynamically with `libC`, and are running Solaris 2.5 or an earlier version of the operating system, you must install the `libC.so.5` patch on your machine: SPARC: 101242-11; x86: 102859-02; PowerPC: 103721-01. Once you have installed this patch, binaries created by the two compilers will be compatible. You can create binary files with either compiler, and link using either compiler, without any compatibility problems.

Compatibility Between C++ 4.1 and C++ 4.2

C++ 4.1 and C++ 4.2 are fully compatible, with one notable exception. If code is compiled with C++ 4.2 and uses the newer cast operations, including runtime type information (RTTI), this code cannot be linked with older versions of `libC`. This is because runtime functions that are missing in the older versions of the C++ library handle `dynamic_cast` and some `typeid()` calls. For code requiring this functionality, you must install the `libC.so.5` patch (SPARC: 101242-11; x86: 102859-02; PowerPC: 103721-01).

Multiple File Extensions

The C++ compiler accepts file extensions other than `.cc`. You can incorporate different file extensions (suffixes) into C++ in two ways: by adding them to the system default makefile or to your makefile.

Suffix Additions to the System Default Makefile

You can add suffixes to C++ by adding them to the system default makefile: `/usr/share/lib/make/make.rules`

Here is an example of how to add suffixes.

1. Become root.
2. If you use Solaris 2.x, `.C` and `.C~` are already in the `SUFFIXES` macro.
3. If they are not already present, add these lines to the end of the system default makefile (indented lines are tabbed):

```
.C:
	$(LINK.cc) -o $@ $< $(LDLIBS)
.C.o:
	$(COMPILE.cc) $(OUTPUT_OPTION) $<
.C.a:
	$(COMPILE.cc) -o $% $<
	$(AR) $(ARFLAGS) $@ $%
	$(RM) $%
```

Note – Since `.c` is supported as a C-language suffix, it is the one suffix that cannot be added to the `SUFFIXES` macro to support C++. Write explicit rules in your own makefile to handle C++ files with a `.c` suffix.

Suffix Additions to Your Makefile

The other method of including different file extensions into C++ is to add them to your makefile.The following example adds `.C` as a valid suffix for C++ files.

1. Add the `SUFFIXES` macro to your makefile:

```
.SUFFIXES: .C .C~
```

This line can be located anywhere in the makefile.

2. Add these lines to your makefile (indented lines are tabbed):

```
.C:
	$(LINK.cc) -o $@ $< $(LDLIBS)
.C.o:
	$(COMPILE.cc) $(OUTPUT_OPTION) $<
.C.a:
	$(COMPILE.cc) -o $% $<
	$(AR) $(ARFLAGS) $@ $%
	$(RM) $%
```

Components

`CC` uses the following components to compile C++ source code to object code.

For SPARC, Intel, and PowerPC Platforms:

- `ccfe` performs preprocessing and compilation.
- `tdb_link` performs template instantiation of out-of-date templates, and invokes the linker.
- `ld` performs link editing.

For SPARC platforms:

- `iropt` optimizes for execution time. This step is optional; any level of optimization invokes this step.
- `cg` performs code generation when optimization is specified.

For Intel and PowerPC platforms:

- `cg386` (*on Intel*) or `cgppc` (*on PowerPC*) prepares intermediate code for code generation.
- `codegen` performs optimizations and code generation.
- `fbe` generates a `.o` file from the `.s` assembly file.

Pragmas

A pragma, also called a *compiler directive*, is a special comment that gives preprocessing instructions to the compiler. Preprocessing lines of the form

`#pragma` *preprocessor-token*

specify implementation-defined actions.

The pragmas discussed in this section are recognized in the compilation system. The compiler ignores unrecognized pragmas.

Note – `#pragmas` in template definition files are ignored.

`#pragma align`

Use `#pragma align` *integer*`(`*variable*`[,`*variable*`]...)`to make the parameter 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.
- 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` *and* `#pragma fini`

Use `#pragma init(`*identifier* `[ ,` *identifier* `] ...)`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. In the case of initializers in a shared object, they 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`.

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

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

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

Use `#pragma pack` to control the layout of structure offsets. *n* is a number, 1, 2, or 4, that specifies the strictest alignment desired for any structure member. If *n* is omitted, members are aligned on their natural boundaries. If you use `#pragma pack` (*n*), be sure to place it after all `#includes`.

#pragma unknown_control_flow

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

Use `#pragma 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 line:

`#pragma weak` *function_name*

defines *function_name* to be a weak symbol. No error messages are generated if the linker cannot find a definition for *function_name*. The line:

`#pragma weak` *function_name1* `=` *function_name2*

defines *function_name1* to be a weak symbol, an alias for the symbol *function_name2*. You must declare *function_name1* and *function_name2* before using them in the pragma. For example:

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

The rules 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.

Options

This section describes the C++ compiler options, arranged alphabetically. These descriptions are also available in the man page, `CC`(1).

In addition, these descriptions are summarized in:

- The *C++ Quick Reference*, a document provided with this compiler
- The `-flags` or `-help` option

Note – In the descriptions are many references to the linker, `ld`. For details about `ld`, see the `ld`(1) man page.

`ccfe` *and* `ld`

The CC driver passes command-line options to the following programs:

- The preprocessor and compiler `ccfe`; see the `CC`(1) man page
- The linker `ld`; see the `ld`(1) man page

The options for `ccfe` and `ld` do not conflict.

`–386`

(Intel) Directs the compiler to generate code for the best performance on the Intel® 80386 microprocessor.

`–486`

(Intel) Directs the compiler to generate code for best performance on the Intel 80486 microprocessor.

`–a`

Prepares object code for coverage analysis, using `tcov`. This option helps you analyze your program at runtime, but is incompatible with `-g`.

This option is the old style of basic block profiling for `tcov`. See `-xprofile=tcov` for information on the new style of profiling, the `tcov`(1) man page, and *Profiling Tools* for more details.

If set at compile time, the `TCOVDIR` environment variable specifies the directory where the `.d` files are located. If this variable is not set, then the `.d` files remain in the same directory as the `.f` files.

The `-xprofile=tcov` and the `-a` options are compatible in a single executable. That is, you can link a program that contains some files that have been compiled with `-xprofile=tcov`, and others that have been compiled with `-a`. You cannot compile a single file with both options.

`–B`*binding*

Specifies whether library bindings for linking are dynamic (shared) or static (nonshared). The values for *binding* are `static` and `dynamic`. `-Bdynamic` is the default. You can use the `-B` option to toggle several times on a command line. All libraries specified after the `–Bstatic` option will be linked statically.

For more information on this option on SPARC, x86 and PowerPC platforms, see the `ld`(1) man page and the Solaris documentation.

`–Bdynamic`

Directs the link editor to look for `lib`*lib*`.so` files. Use this option if you want shared library bindings for linking. If the `lib`*lib*`.so` files are not found, it looks for `lib`*lib*`.a` files. By default, the `CC` driver passes the following `-l` options to `ld`:

```
-lC -lC_mtstubs -lm -lw -lcx -lc
```

For more details on `-l`, refer to “-l*lib*” on page 31.

To link some of these libraries statically, use `-nolib`, as described in “–nolib” on page 34. To link `libC` statically, you can also use the `–staticlib=libC` option.

–Bstatic

Directs the link editor to look *only* for files named `lib`*lib*`.a`. The `.a` suffix indicates that the file is static, that is, nonshared. Use this option if you want nonshared library bindings for linking. On Solaris, this option and its arguments are passed to the linker, `ld`.

–c

Directs the `CC` driver to suppress linking with `ld` and produces a `.o` file for each source file. If you specify only one source file on the command-line, then you can explicitly name the object file with the `-o` option. For example:

- If you enter **`CC -c x.cc`**, the object file, `x.o`, is generated.
- If you enter **`CC -c x.cc -o y.o`**, the object file, `y.o`, is generated.

See also "–o filename" on page 39.

–cg[89|92]

(SPARC) Specifies the code generator for floating-point hardware in SPARC systems released in 1989 or 1992. Use the `fpversion` command to determine which floating-point hardware you have. If you compile one procedure of a program with this option, it does not mean that you must compile *all* the procedures of that program in the same way.

–cg89

(SPARC)–`cg89` expands to:
`-xarch=v7 -xchip=old -xcache=64/32/1.`

–cg92

(SPARC) –`cg92` expands to:
`-xarch=v8 -xchip=super -xcache=16/32/4:1024/32/1.`

`+d`

Prevents the compiler from expanding inline functions. This option is turned on when you specify `-g`, the debugging option.

The debugging option, `-g0`, does not turn on `+d`. See "–g0" on page 28.

`-D`*name*`[`=*def*`]`

Defines a macro symbol *name* to the preprocessor. Doing so is equivalent to including a `#define` directive at the beginning of the source. If you do not use the argument =*def*, *name* is defined as `1`. You can use multiple `-D` options.

The following values are predefined:

- `__BUILTIN_VA_ARG_INCR` (for the `__builtin_alloca`, `__builtin_va_alist`, and `__builtin_va_arg_incr` keywords in `varargs.h`, `stdarg.h`, and `sys/varargs.h`)
- `__cplusplus`
- `__DATE__`
- `__FILE__`
- `__LINE__`
- `__STDC__`
- `__sun`
- `sun`
- `__SUNPRO_CC=0x420`
- `__SVR4`
- `__TIME__`
- `__`<*uname* `-s`>`_`<*uname* `-r`> (replaces invalid characters with underscores, as in `-D__SunOS_5_3; -D__SunOS_5_4;`
- `__unix`
- `unix`
- `_WCHAR_T_`

SPARC only:
- `__sparc`
- `sparc`

x86 only:
- `__i386`
- `i386`

PowerPC only:

- `__ppc`

The `sparc`, `unix`, `sun`, and `i386` macros are not defined if `+p` is used. The value of `__SUNPRO_CC` indicates the release number of the compiler. You can use these values in such preprocessor conditionals as `#ifdef`.

`–d[y|n]`

`-dy` specifies dynamic linking, which is the default, in the link editor.

`-dn` specifies static linking in the link editor.

This option and its arguments are passed to `ld`.

`–dalign`

(SPARC) Generates `double-word load` and `store` instructions whenever possible for improved performance. This option assumes that all `double` type data are `double-word` aligned. If you compile one unit with `–dalign`, compile all units of a program with `–dalign`, or you may get unexpected results.

`–dryrun`

Displays what options the driver has passed to the compiler. This option directs the driver `CC` to show, but not execute, the commands constructed by the compilation driver.

`–E`

Directs the `CC` driver to run only the preprocessor on C++ source files, and to send the result to `stdout` (standard output). No compilation is done; no `.o` files are generated.

`+e[0|1]`

Controls virtual table generation, as follows:

- `+e0` suppresses the generation of virtual tables and creates external references to those that are needed.
- `+e1` creates virtual tables for all defined classes with virtual functions. When you compile with this option, also use the `-noex` option; otherwise, the compiler generates virtual tables for internal types used in exception handling.

See also Appendix A, "Migration Guide" on page 169.

–fast

Selects a combination of compilation options for optimum execution speed. This option provides near maximum performance for most applications by choosing the following compilation options:

Table 2-1 Compilation Options Selected by `–fast`

Option	SPARC	x86	PowerPC
`–dalign`	X		
`–fns`	X	X	X
`–fsimple`	X		
`–ftrap=%none`	X	X	X
`–libmil`	X	X	X
`–nofstore`		X	
`–O4`	X	X	X
`–xlibmopt`	X	X	
`–xtarget=native`	X	X	X

The code generation option, optimization level, and use of inline template files can be overridden by subsequent flag switches. For example, although the optimization level set by `–fast` is `–O4`, if you specify `-fast -O3,` the optimization level becomes `-O3`. The optimization level that you specify will override a previously set optimization level.

Do not use this option for programs that depend on IEEE standard floating-point exception handling; different numerical results, premature program termination, or unexpected `SIGFPE` signals may occur.

Note – The criteria for the `-fast` option vary with the C, C++, Fortran 77, and Pascal compilers. Please see the appropriate documentation for the specifics.

The `-fast` option includes `-fns -ftrap=%none`; that is, this option turns off all trapping. In previous SPARC releases, the `-fast` macro option included `-fnonstd;` now it does not.

`–features=`*a*

Enables/disables various C++ language features. *a* must be one or more of: [no%]`anachronisms`, [no%]`castop`, [no%]`rtti`

The following table shows the values of *a*.

Value	**Meaning**
`[no%]anachronisms`	[Do not] Allow anachronistic constructs
`[no%]castop`	[Do not] Allow new-style casts (dynamic or otherwise)
`[no%]rtti`	[Do not] Allow RTTI (`dynamic_cast <>` and `typeid`)

`–flags`

Displays a brief description of each compiler option. Also displayed are phone numbers to call for additional information on Sun products and technical support, and instructions on how to send comments on Sun products.

`–fnonstd`

Causes nonstandard initialization of floating-point arithmetic hardware. In addition, the `–fnonstd` option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These results are converted into SIGFPE signals; if the program has no SIGFPE handler, it terminates with a memory dump. See the *Numerical Computation Guide* for more information.

By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual.

This option is a synonym for `–fns –ftrap=common`.

`–fns`

Turns on the nonstandard floating-point mode. The default is the standard floating-point mode.

If you compile one routine with `-fns`, then compile all routines of the program with the `–fns` option; otherwise, you can get unexpected results.

`–fprecision=`*p*

(Intel) Sets floating-point rounding precision mode.

p must be one of: `single`, `double`, `extended`. The following table shows the values of *p*.

Value	**Meaning**
`single`	Rounds to an IEEE single-precision value
`double`	Rounds to an IEEE double-precision value
`extended`	Rounds to the maximum precision available

`–fround=`*r*

Sets the IEEE 754 rounding mode.

r must be one of `nearest`, `tozero`, `negative`, or `positive`.

The default is `-fround=nearest`.

This option sets the IEEE 754 rounding mode that:

- Can be used by the compiler in evaluating constant expressions
- Is established at runtime during the program initialization

The meanings are the same as those for the `ieee_flags` subroutine.

If you compile one routine with `-fround=`*r*, compile all routines of the program with the same `–fround=`*r* option; otherwise, you can get unexpected results.

`–fsimple[=`*n*`]`

Allows the optimizer to make simplifying assumptions concerning floating-point arithmetic.

If *n* is present, it must be 0, 1 or 2. The defaults are:

- Without `-fsimple[=`*n*`]`, the compiler uses `-fsimple=0`
- With only `-fsimple`, no ”=*n*”, the compiler uses `-fsimple=1`

`-fsimple=0`

Permits no simplifying assumptions. Preserves strict IEEE 754 conformance.

`-fsimple=1`

Allows conservative simplification. The resulting code does not strictly conform to IEEE 754, but numeric results of most programs are unchanged.

With `-fsimple=1`, the optimizer can assume the following:

- IEEE754 default rounding/trapping modes do not change after process initialization
- Computation producing no visible result other than potential floating point exceptions may be deleted
- Computation with Infinity or NaNs as operands needs to propagate NaNs to their results; e.g., x*0 may be replaced by 0.
- Computations do not depend on sign of zero

With `-fsimple=1`, the optimizer is not allowed to optimize completely without regard to roundoff or exceptions. In particular, a floating point computation cannot be replaced by one that produces different results with rounding modes held constant at runtime. `-fast` implies `-fsimple=1`.

`-fsimple=2`

Permits aggressive floating point optimization that may cause many programs to produce different numeric results due to changes in rounding. For example, permits the optimizer to replace all computations of `x/y` in a given loop with `x*z`, where `x/y` is guaranteed to be evaluated at least once in the loop, `z=1/y`, and the values of `y` and `z` are known to have constant values during execution of the loop.

–fstore

Causes the compiler to convert the value of a floating-point expression or function to the type on the left side of an assignment—when that expression or function is assigned to a variable, or when the expression is cast to a shorter floating-point type rather than leaving the value in a register. Due to roundoffs and truncation, the results may be different from those that are generated from the register values. This is the default mode.

To turn off this option, use the `-nofstore` option.

–ftrap=*t*

Sets the IEEE 754 trapping mode.

t is a comma-separated list that consists of one or more of the following: `%all`, `%none`, `common`, [no%]`invalid`, [no%]`overflow`, [no%]`underflow`, [no%]`division`, [no%]`inexact`.

The default is `-ftrap=%none`.

This option sets the IEEE 754 trapping modes that are established at program initialization. Processing is left to right. The common exceptions, by definition, are `invalid`, `division` (by zero), and `overflow`.

For example: `-ftrap=%all,no%inexact` means to set all traps except `inexact`.

The meanings are the same as for the `ieee_flags` subroutine, except that:

- `%all` turns on all the trapping modes.
- `%none`, the default, turns off all trapping modes.
- A `no%` prefix turns off that specific trapping mode.

If you compile one routine with `-ftrap=`*t*, compile all routines of the program with the same `-ftrap=`*t* option; otherwise, you can get unexpected results.

–G

Instructs the linker to build a shared library; see the `ld`(1) man page and the *C++ Library Reference*. All source files specified in the command line are compiled with `-pic`.

Use this option when building shared libraries of templates. Any generated templates are then automatically included in the library.

The following text options are passed to `ld`:

- `-dy`
- `-G`
- `-ztext`

–g

Instructs both the compiler and the linker to prepare the file or program for debugging. The tasks include:

- Producing more detailed information, known as *stabs*, in the symbol table of the object files and the executable
- Producing some “helper functions,” which the debugger can call to implement some of its features
- Disabling the inline generation of functions; that is, using this option implies the `+d` option as well
- Disabling certain levels of optimization; you can use this option along with `-O` to get the optimization level that you desire

This option makes `-xildon` the default incremental linker option in order to speed up the compile-edit-debug cycle. See ”-xildon” and “-xildoff.” Invokes `ild` in place of `ld` unless any of the following are true:

- The `-G` option is present
- The `-xildoff` option is present
- Any source files are named on the command line

See also the descriptions for “–g0” and “+d,” as well as the `ld`(1) man page. The `dbx` *User’s Guide* provides details about stabs and “lazy stabs.”

–g0

Instructs the compiler to prepare the file or program for debugging, but *not* to disable inlining. This option is the same as `–g`, except that `+d` is not enabled.

See also "+d" on page 20.

–H

Prints, one per line, the path name of each `#include` file included during the current compilation on the standard error output (`stderr`).

–help

Same as `-flags`, as described in "–flags" on page 23.

–h*name*

Names a shared dynamic library and provides a way to have versions of a shared dynamic library.

This is a loader option, passed to `ld`. In general, the name after `-h` should be exactly the same as the one after `-o`. A space between the `-h` and *name* is optional.

The compile-time loader assigns the specified name to the shared dynamic library you are creating. It records the name in the library file as the intrinsic name of the library. If there is no `-h`*name* option, then no intrinsic name is recorded in the library file.

Every executable file has a list of needed shared library files. When the runtime linker links the library into an executable file, the linker copies the intrinsic name from the library into that list of needed shared library files. If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead. This command line is an example:

```
% CC -G -o libx.so.1 -h libx.so.1 a.o b.o c.o
```

–i

Tells the linker to ignore any LD_LIBRARY_PATH setting.

–I*pathname*

Adds *pathname* to the list of directories that are searched for `#include` files with relative file names—those that do not begin with a slash. The preprocessor searches for `#include` files in this order:

1. For includes of the form `#include "foo.h"` (where quotation marks are used), the directory containing the source file is searched.

   For includes of the form `#include <foo.h>` (where angle brackets are used), the directory containing the source file is *not* searched
2. In the directories named with `-I` options, if any
3. In the standard directory for C++ header files:

```
/opt/SUNWspro/SC4.2/include/CC
```

4. In the standard directory for ANSI C header files:

```
/opt/SUNWspro/SC4.2/include/cc
```

5. In `/usr/include`.

Note – If `–pti`*path* is not used, the compiler looks for template files in `–I`*path*. Use `–I`*path* instead of `–pti`*path*.

–inline=*rlst*

Inlines the routines that you specify in the *rlst* list—a comma-separated list of functions and subroutines. This option is used by the optimizer.

Note – This option does not affect C++ inline functions and is not related to the `+d` option.

If a function specified in the list is not declared as `extern “C”`, the function name should be mangled. You can use the `nm` command on the executable file to find the mangled function names. For functions declared as `extern "C"`, the names are not mangled by the compiler.

If you compile with the flag `-O3`, using this option can increase optimization by restricting inlining to only those routines in the *rlst* list.

If you compile with the flag `-O4`, the compiler tries to inline all user-written subroutines and functions.

A routine is not inlined if any of the following conditions apply—no warning is issued.

- Optimization is less than `-O3`.
- A routine cannot be found.
- The compiler does not consider inlining the routine advantageous or safe.
- The source for the routine is not in the file being compiled.

`–instances=`*a*

Instantiates templates with the linkage you specify within the current object.

a must be one of: `static`, `extern`, `global`. These linkages are mutually exclusive. The following table shows the values of *a*.

Value	**Meaning**
`extern`	Instantiates generated templates within the template database, reinstantiating only as necessary. The current compilation unit references the instantiation by extern symbols. (Default)
`static`	Instantiates generated templates in the current compilation unit. Generated templates are then made static. The only restriction is that all template definitions must be located in the source or headers you are compiling.
`global`	Instantiates generated templates in the current compilation unit. Generated templates are then made global. The only restriction is that all template definitions must be located in the source or headers you are compiling.

–keeptmp

Retains the temporary files that are created during compilation. Along with the `-v` option, this option is useful for debugging, especially when you have template functions or classes.

–KPIC

Same as the `-PIC` option, as described in "–PIC" on page 40.

–Kpic

Same as the `-pic` option, as described in "–pic" on page 40.

–L*dir*

Adds *dir* to the list of directories to be searched by the linker for libraries that contain object-library routines during the link step with `ld`. The directory, *dir*, is searched before compiler-provided directories.

C++ Library Reference and *Tools.h++ Class Library Reference* contain further information on some of the C++ libraries.

–l*lib*

Specifies additional libraries for linking with object files. This option behaves like the `-l` option when it is used with `CC` or `ld`. Normal libraries have names like `lib`*something*`.a`, where the `lib` and `.a` parts are required. You can specify the *something* part with this option. Put as many libraries as you want on a single command line; they are searched in the order specified with `–L`.

Use this option after your file name.

–libmieee

Causes `libm` to return values in the spirit of IEEE 754. The default behavior of `libm` is SVID-compliant.

`–libmil`

Inlines some library routines for faster execution. This option selects the best assembly language inline templates for the floating-point option and platform on your system.

Note – This option does not affect C++ inline functions.

`–library=`*l*

Determines C++ library use.

If a library is specified with `-library`, the proper `–I` (as well as `–L`, `–Y P`, `–R` and `–l`) paths are passed to the compiler in order to use a particular library. During the link phase, the correct libraries will be presented to the linker.

l must be one of: `[no%]rwtools7`, `[no%]rwtools6`, `[no%]libC`, `[no%]libm`, `[no%]complex`, `%all`, `%none`. The following table shows the values of *l*.

Value	**Meaning**
`[no%]rwtools7`	[Do not] Use Tools.h++ v 7
`[no%]rwtools6`	[Do not] Use Tools.h++ v 6
`[no%]libC`	[Do not] Use `libC`
`[no%]libm`	[Do not] Use `libm`
`[no%]complex`	[Do not] Use `libcomplex`
`%all`	Use all libraries, in the order: `rwtools7`, `complex`, `libC`, `libm`
`%none`	Use no C++ libraries

If selected, libraries are linked in the order: `rwtools7`, `rwtools6`, `complex`, `libC`, `libm`. The default is `-library=%none`, `libC`, `libm`.

See also “–staticlib=l” on page 43.

–migration

Displays the contents of the *C++ Migration Guide*, which contains information about incompatibilities between C++ 3.0, based on `Cfront`, and the current compiler.

Move through the contents of the file using the command specified by the environment variable, `PAGER`. If this environment variable is not set, the default paging command is `more`.

–misalign

(SPARC) (PowerPC) Permits misaligned data, which would otherwise generate an error, in memory, as shown in the following code:

```
char b[100];
int f(int *ar){
return   *(int *)(b +2) +  *ar;
}
```

Thus, very conservative loads and stores must be used for the data, that is, one byte at a time. Using this option can cause significant degradation in performance when you run the program.

If possible, do not link aligned and misaligned parts of the program.

–mt

Compiles and links a multithreaded program and passes `-D_REENTRANT` to the preprocessor. The following command is passed to `ld`:

`-lC -lm -lw -lthread -lcx -lc`

instead of:

`-lC -lm -lC_mtstubs -lw -lcx -lc`

Note – To ensure proper library linking order, use this option rather than `-lthread`, to link with `libthread`.

–native

Chooses the correct code generator option. This option directs the compiler to generate code targeted for the machine that is doing the compilation.

Native floating-point—use what is best for this machine.

This option is a synonym for `-xtarget=native`.

The `-fast` macro includes `-native` in its expansion.

–noex

Instructs the compiler not to generate code that supports C++ exceptions.

–nofstore

(Intel) Does not convert the value of a floating-point expression or function to the type on the left side of an assignment when that expression or function is assigned to a variable, or is cast to a shorter floating-point type; rather, leaves the value in a register.

See also "–fstore" on page 26.

–nolib

Does not link with any system or library by default; that is, no `-l` options are passed to `ld`. You can then link some of the default libraries statically instead of dynamically.

You must pass all `-l` options explicitly. The `CC` driver normally passes the following options to `ld`:

```
-lC -lm -lC_mtstubs -lw -lcx -lc
```

You can link `libC`, `libm`, and `libw` statically, and link `libc` dynamically, as follows:

```
% CC test.c -nolib -Bstatic -lC -lm -lC_mtstubs -lw -lcx \
 -Bdynamic -lc
```

The order of the `-l` options is significant. The `-lC`, `-lm`, `-lC_mtstubs`, `-lw`, and `-lcx` options must appear before `-lc`.

See also “–library=l” on page 32 and “–staticlib=l” on page 43.

(PowerPC only) In this release, all C++ programs will be linked with `-labi`, which the driver adds to the link line. If you choose to use `-nolib`, you must manually link with `-labi`. If you do not link with `-labi`, the following three entry points required for exception handling will not be defined, and the program will not link:

`_add_module_tags`

`_delete_module_tags`

`_tag_lookup_pc`

In the next OS release, these entry points will appear in `libc`, and `-labi` will cease to exist.

As above, the order of the options is significant. The `-lC` option must appear before `-labi.`

–nolibmil

Resets `-fast` so that it does *not* include inline templates. Use this option after the `-fast` option, as in:

```
% CC –fast –nolibmil
```

–noqueue

If no license is available, returns without queuing your request and without compiling. A nonzero status is returned for testing makefiles.

–norunpath

Does not build the path for shared libraries into the executable. If an executable file uses shared libraries, then the compiler normally builds in a path that points the runtime linker to those shared libraries. To do so, the compiler passes the `–R` option to `ld`. The path depends on the directory where you have installed the compiler.

This option is helpful if you have installed the compiler in some nonstandard location, and you ship an executable file to your customers, who need not work with that nonstandard location.

If you use any shared libraries under the compiler installed area (default location `/opt/SUNWspro/lib`) and you also use `–norunpath`, then you should either use the `–R` option at link time or set the environment variable `LD_LIBRARY_PATH` at run time to specify the location of the shared libraries. This will allow the runtime linker to find the shared libraries.

`–O`*level*

Optimizes the execution time. Omitting *level* is equivalent to using level `-O2`.

There are five levels that you can use with `-O`. The following sections describe how they differ for SPARC systems, for x86, and for all systems.

For SPARC Systems

`–O1`: Does only the minimum amount of optimization (peephole), which is postpass, assembly-level optimization. Do not use `-O1` unless using `-O2` or `-O3` results in excessive compilation time, or you are running out of swap space.

`–O2`: Does basic local and global optimization, which includes:

- Induction-variable elimination
- Local and global common-subexpression elimination
- Algebraic simplification
- Copy propagation
- Constant propagation
- Loop-invariant optimization
- Register allocation
- Basic block merging
- Tail recursion elimination
- Dead-code elimination
- Tail-call elimination
- Complex-expression expansion

This level does not optimize references or definitions for external or indirect variables. Do not use `–O2` unless using `–O3` results in excessive compilation time, or you are running out of swap space. In general, this level results in minimum code size.

`–O3:` Also optimizes references and definitions for external variables in addition to optimizations performed at the `-O2` level. This level does not trace the effects of pointer assignments. Do not use it when compiling either device drivers, or programs that modify external variables from within signal handlers. In general, this level results in increased code size.

`–O4:` Does automatic inlining of functions contained in the same file in addition to performing `-O3` optimizations. This automatic inlining usually improves execution speed, but sometimes makes it worse. In general, this level results in increased code size.

–`O5`: Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time. Optimization at this level is more likely to improve performance if it is done with profile feedback. See"–xprofile=p" on page 56.

For Intel and PowerPC Systems

`–O1:` Preloads arguments from memory; causes cross jumping (tail merging), as well as the single pass of the default optimization.

`–O2`: Schedules both high- and low-level instructions and performs improved spill analysis, loop memory-reference elimination, register lifetime analysis, enhanced register allocation, global common subexpression elimination, as well as the optimization done by level 1.

`–O3`: Performs loop strength reduction and inlining, as well as the optimization done by level 2.

`–O4`: Performs architecture-specific optimization, as well as the optimization done by level 3.

`–O5`: Generates the highest level of optimization. Uses optimization algorithms that take more compilation time or that do not have as high a certainty of improving execution time.

For All Systems

For most programs:

- Level `–O5` is faster than `–O4`
- Level `–O4` is faster than `–O3`
- Level `-O3` is faster than `–O2`
- Level `-O2` is faster than `–O1`

In a few cases, `–O2` may perform better than the others, and `–O3` may outperform `–O4`. Try compiling with each level to see if you have one of these rare cases.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization. The optimizer then resumes subsequent procedures at the original level specified in the `–O` option.

If you optimize at `–O3` or `–O4` with very large procedures (thousands of lines of code in a single procedure) the optimizer may require an unreasonable amount of memory. In such cases, machine performance may be degraded.

To prevent this degradation from taking place, use the `limit` command to limit the amount of virtual memory available to a single process; see the `csh`(1) man page. For example, to limit virtual memory to 16 megabytes:

```
% limit datasize 16M
```

This command causes the optimizer to try to recover if it reaches 16 megabytes of data space.

This limit cannot be greater than the total available swap space of the machine, and the limit should be small enough to permit normal use of the machine while a large compilation is in progress.

For example, on a machine with 32 megabytes of swap space, the `limit datasize 16M` command ensures that a single compilation never consumes more than half of the machine swap space.

The best setting for data size depends on the degree of optimization requested, the amount of real memory, and virtual memory available.

- To find the actual swap space, type: **`swap -l`**
- To find the actual real memory, type: **`dmesg | grep mem`**

`–o` *filename*

Sets the name of the output file (with the suffix, `.o`) or the executable file to *filename*. The file name must have the appropriate suffix for the type of file to be produced by the compilation. It cannot be the same file as the source file, since the `CC` driver does not overwrite the source file.

`+p`

Use `–features=no%anachronisms`.

`–P`

Runs a source file through the preprocessor, which outputs a file with a `.i` suffix. This option does not include preprocessor-type line number information in the output.

`–p`

Prepares the object code to collect data for profiling with `prof`. This option invokes a runtime recording mechanism that produces a `mon.out` file at normal termination.

You can also perform this task with the Analyzer. Refer to the `analyzer`(1) man page.

`–pentium`

(Intel) Use `–xtarget=pentium`.

`–pg`

Prepares the object code to collect data for profiling with `gprof`. This option invokes a runtime recording mechanism that produces a `gmon.out` file at normal termination.

You can also perform this task with the Analyzer. Refer to the `analyzer`(1) man page.

–PIC

(Intel) Produces position-independent code.

Same as `-pic`.

(SPARC) (PowerPC) Similar to `-pic`, but the global offset table spans the range of 32-bit addresses in those rare cases where there are too many global data objects for `-pic`.

–pic

Produces position-independent code. Use this option to compile source files when building a shared library.

Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8 Kbytes on SPARC processors.

–pta

Use `–template=wholeclass.` See also "`-template=`*w*" on page 44.

–pti*path*

Specifies an additional search directory for template source.

This option is an alternative to the normal search path set by `–I`*pathname*. If the `–pti`*path* flag is used, the compiler looks for template definition files on this path and ignores the `–I`*pathname* flag. It is recommended that you use the `–I`*pathname* flag instead of `–pti`*path*.

–pto

Use `–instances=static.` See also "`-instances=`*a*" on page 30.

`–ptr`*database-path*

Specifies the directory of primary and secondary build repositories. The template database is named `Templates.DB`; *database-path* is the directory containing the database, but does not include the name itself.

For example:

- `-ptr/tmp/Foo` specifies the database `/tmp/Foo/Templates.DB`
- `-ptr.` specifies the database `./Templates.DB`

You can use multiple `-ptr` options. However, all repositories specified, except the first one, are read-only; no templates are instantiated into these directories.

If you omit this option, a default database, `./Templates.DB`, is created for you. To avoid confusion when multiple databases are involved, use `-ptr.` as the first entry on the command line.

`–ptv`

Use `–verbose=template`. See also "`-verbose=`*v*" on page 46.

`–Qoption|-qoption` *prog opt*

Passes the option, *opt*, to the phase, *prog*. Refer to "Components" on page 13 for more information on compiler phases.

Table 2-2 shows the possible values for *prog*:

Table 2-2 Compiler Phases

SPARC Architecture	**x86 Architectures**
`ccfe`	`ccfe`
`iropt`	`cg386`
`cg`	`codegen`
`tdb_link`	`tdb_link`
`ld`	`ld`

To pass more than one option, specify them in order as a comma-separated list. In the following command line, when `ld` is invoked by the `CC` driver, `-Qoption` passes the `-i` and `-m` options to `ld`:

```
% CC -Qoption ld -i,-m test.c
```

–qp

Use `-p`.

–Qproduce|-qproduce *sourcetype*

Causes the `CC` driver to produce source code of the type *sourcetype*. Source code types are shown in Table 2-3.

Table 2-3 Source Code Types

`.i`	Preprocessed C++ source from `ccfe`
`.o`	Object file from `cg`, the code generator
`.s`	Assembler source from `cg`

–R *pathname*

Passes a colon-separated list of search paths to the linker for library searching during runtime linking.

Multiple instances of `-R`*pathname* are concatenated, with each *pathname* separated by a colon. Without this option specified, the compiler passes a default search path to the linker:

`/opt/SUNWspro/lib` (for standard installs)

`<install_path>/lib` (for non-standard installs into `<install_path>`)

With `-R`*pathname* specified, the listed library search paths are added to the default paths passed to the linker and built into the object program.

`(The LD_RUN_PATH` environment variable is always ignored.)

–readme

Displays the contents of the `README` file, paged by the command specified by the environment variable, `PAGER`. If `PAGER` is not set, the default paging command is `more`.

For a description of the contents of this file, see “README file” on page xxiii.

–S

Causes the `CC` driver to compile the program and output an assembly source file, but not assemble the program. The assembly source file is named with a `.s` suffix.

–s

Removes all symbol information from output executable files. This option is passed to `ld`.

–sb

Causes the `CC` driver to generate extra symbol table information in a SourceBrowser database in the `.sb` directory for the `sbrowser` program.

–sbfast

Runs only the `ccfe` phase to generate extra symbol table information in a SourceBrowser database in the `.sb` directory for the `sbrowser` program. No object file is generated.

–staticlib=*l*

Determines whether C++ libraries are linked statically.

If a library is specified with both `-library` and with `-staticlib`, that library is linked statically. If the library is specified with `–staticlib`, but not with `-library`, `–staticlib` is ignored.The default for `-staticlib` is `staticlib=%none`.

l must be one or more of [no%]`rwtools6`, [no%]`rwtools7`, [no%]`libC`,[no%]`libm`,[no%]`complex`, `%all`, `%none`.

The following table shows the values of *l*.

Value	**Meaning**
`[no%]rwtools6`	Tools.h++ v 6 [not] linked statically
`[no%]rwtools7`	Tools.h++ v 7 [not] linked statically
`[no%]libC`	`libC` [not] linked statically
`[no%]libm`	`libm` [not] linked statically
`[no%]complex`	`libcomplex` [not] linked statically
`%all`	All libraries specified in the `-library` option linked statically. If no libraries are specified using `-library`, then `%all` means link the libraries `libC` and `libm` statically.
`%none`	Dynamic linking

For example, you can link libC statically as follows:

```
% CC test.c -staticlib=libC
```

`–temp=`*dir*

Sets the name of the directory for temporary files, generated during the compilation process, to *dir*. See also "–keeptmp" on page 31.

`–template=`*w*

Enables / disables various template options.

The following table shows the values of *w*.

Value	Meaning
`[no%]wholeclass`	Directs the compiler [not] to instantiate a whole template class, rather than only those functions that are used. This option creates a `.o` file for each member of a class. You must reference at least one member of the class, otherwise, the compiler does not instantiate any members for the class.

`–time`

Causes the `CC` driver to report execution times for various compilation passes.

`–U`*name*

Removes any initial definition of the macro symbol *name*. This option:

- Is processed by `ccfe`
- Is the inverse of the `-D` option, as described in "–Dname[=def]" on page 20.

You can specify multiple `-U` options on the command line.

`–unroll=`*n*

Specifies whether or not the compiler optimizes (unrolls) loops.

- When *n* is `1`, it is a suggestion to the compiler to not unroll loops.
- When *n* is an integer greater than `1`, `-unroll=`*n* causes the compiler to unroll loops *n* times.

`–V`

Use `–verbose=version.`

`–v`

Use `–verbose=`diags.

`–verbose=`*v*

Controls verbosity during compilation.

v must be one or more of: `template`, `diags`, `version`.

You may specify more than one option, for example, `–verbose=template, diags`

The following table shows the values of *v*.

Value	**Meaning**
`template`	Turns on the verbose mode, sometimes called the verify mode. The verbose mode displays each phase of instantiation as it occurs during compilation. Use this option if you are new to templates.
`diags`	Prints the command line for each compilation pass
`version`	Directs the CC driver to print the names and version numbers of the programs it invokes.

`+w`

Generates additional warnings about questionable constructs that are:

- Nonportable
- Likely to be mistakes
- Inefficient

By default, the compiler warns about constructs that are almost certainly problems.

`–w`

Causes the `CC` driver to *not* print warning messages from the compiler.

`–xa`

Same as the `-a` option, as described in "–a" on page 18.

–xar

Creates archive libraries.

When compiling a C++ archive that uses templates, it is necessary in most cases to include in the archive those template functions that are instantiated in the template database. Using this option automatically adds those templates to the archive as needed.

For example:

```
CC -xar -O libmain.a a.o b.o c.o
```

archives the template functions contained in the library and object files.

–xarch=*a*

Limits the set of instructions the compiler may use.

Target architectures specified by keyword *a* are:

Table 2-4 The `-xarch` Architecture Keywords

On SPARC:	`generic, v7, v8a, v8, v8plus, v8plusa`
On PowerPC:	`generic, ppc, ppc_nofma`
On Intel:	`generic, 386, pentium_pro`

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option limits the instructions generated to those appropriate to the specified architecture, and *allows* the specified set of instructions. The option does not guarantee an instruction will be used; however, under optimization, it is usually used.

If this option is used with optimization, the appropriate choice can provide good performance of the executable on the specified architecture. An inappropriate choice can result in serious degradation of performance.

SPARC architectures `v7`, `v8a`, and `v8` are all binary compatible. `v8plus` and `v8plusa` are binary compatible with each other, and they are forward but not backward compatible.

For any particular choice, the generated executable can run much more slowly on earlier architectures.

Table 2-5 The `-xarch` Values

Value	**Meaning**
`generic`	Gets good performance on most SPARCs, major degradation on none. This is the default. This option uses the best instruction set for good performance on most SPARC processors without major performance degradation on any of them. With each new release, this best instruction set will be adjusted, if appropriate.
`v7`	Limits instruction set to V7 architecture. This option uses the best instruction set for good performance on the V7 architecture, but without the quad-precision floating-point instructions. This is equivalent to using the best instruction set for good performance on the V8 architecture, but *without* the following instructions: The quad-precision floating-point instructions The integer `mul` and `div` instructions The `fsmuld` instruction Examples: SPARCstation 1, SPARCstation 2
`v8a`	Limits instruction set to the V8a version of the V8 architecture. By definition, V8a means the V8 architecture, but without: The quad-precision floating-point instructions The `fsmuld` instruction This option uses the best instruction set for good performance on the V8a architecture. Example: Any machine based on MicroSPARC I chip architecture

Table 2-5 The `-xarch` Values *(Continued)*

Value	Meaning
`v8`	Limits instruction set to V8 architecture. This option uses the best instruction set for good performance on the V8 architecture, but without quad-precision floating-point instructions. Example: SPARCstation 10
`v8plus`	Limits instruction set to the V8plus version of the V9 architecture. By definition, V8plus means the V9 architecture, except: Without the quad-precision floating-point instructions Limited to the 32-bit subset defined by the V8+ specification Without the VIS instructions This option uses the best instruction set for good performance on the V8plus chip architecture. In V8plus, a system with the 64-bit registers of V9 runs in 32-bit addressing mode, but the upper 32 bits of the i and l registers must not affect program results. Example: Any machine based on UltraSPARC chip architecture. Use of this option also causes the `.o` file to be marked as a V8plus binary. Such files will not run on a V7 or V8 machine.
`v8plusa`	Limits instruction set to the V8plusa architecture variation. By definition, V8plusa means the V8plus architecture, plus: The UltraSPARC-specific instructions The VIS instructions This option uses the best instruction set for good performance on the UltraSPARC™ architecture, but limited to the 32-bit subset defined by the V8plus specification. Example: Any machine based on UltraSPARC chip architecture Use of this option also causes the `.o` file to be marked as a Sun-specific V8plus binary. Such files will not run on a V7 or V8 machine.

For PowerPC:

`generic` and `ppc` are equivalent in this release and direct the compiler to produce code for the PowerPC 603 and 604 instruction set.

`ppc_nofma` is the same as `ppc` except that the compiler will not issue the "fused multiply-add" instruction.

For Intel:

`generic` and `386` are equivalent in this release.

`pentium_pro` directs the compiler to issue instructions for the Intel PentiumPro chip.

`–xcache=`*c*

(SPARC) Defines the cache properties that the optimizer can use. It does not guarantee that any particular cache property is used.

c must be one of the following:

- `generic`
- *s1*/*l1*/*a1*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*
- *s1*/*l1*/*a1*:*s2*/*l2*/*a2*:*s3*/*l3*/*a3*

That is, `-xcache=`{`generic` |*s1*/*l1*/*a1*[:*s2*/*l2*/*a2*[:*s3*/*l3*/*a3*]]}, where the *si*/*li*/*ai* are defined as follows:

si	The size of the data cache at level *i*, in kilobytes
li	The line size of the data cache at level *i*, in bytes
ai	The associativity of the data cache at level *i*

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

.

Table 2-6 The `-xcache` Values

Value	Meaning
`generic`	Defines the cache properties for good performance on most SPARC processors. This is the default value that directs the compiler to use cache properties for good performance on most SPARC processors, without major performance degradation on any of them.
s1/*l1*/*a1*	Defines level 1 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*	Defines levels 1 and 2 cache properties.
s1/*l1*/*a1*:*s2*/*l2*/*a2*: *s3*/*l3*/*a3*	Defines levels 1, 2, and 3 cache properties

Example: `-xcache=16/32/4:1024/32/1` specifies the following:

Level 1 cache has:	Level 2 cache has:
16 Kbytes	1024 Kbytes
32 bytes line size	32 bytes line size
4-way associativity	Direct mapping associativity

`–xcg89`

(SPARC) Same as the `-cg89` option. See *"–cg[89|92]"* on page 19.

`–xcg92`

(SPARC) Same as the `-cg92` option. See *"–cg[89|92]"* on page 19.

`–xchip=`*c*

(SPARC) Specifies the target processor for use by the optimizer.

c must be one of the following:

On SPARC: `generic`, `old`, `super`, `super2`, `micro`, `micro2`, `hyper`, `hyper2`, `powerup`, `ultra`

On PowerPC: `generic, 603, 604`

On Intel: `386, 486, pentium, pentium_pro.`

Although this option can be used alone, it is part of the expansion of the `-xtarget` option; its *primary use* is to override a value supplied by the `-xtarget` option.

This option specifies timing properties by specifying the target processor.

This option affects:

- The ordering of instructions, that is, scheduling
- The way the compiler uses branches
- The instructions to use in cases where semantically equivalent alternatives are available

Table 2-7 The `-xchip` Values

	Value	**Meaning**
SPARC:	`generic`	Uses timing properties for good performance on most SPARC processors. This is the default value that directs the compiler to use the best timing properties for good performance on most SPARC processors, without major performance degradation on any of them.
	`old`	Uses timing properties of pre-SuperSPARC™ processors.
	`super`	Uses timing properties of the SuperSPARC chip.
	`super2`	Uses timing properties of the SuperSPARC II chip.
	`micro`	Uses timing properties of the MicroSPARC™ chip.
	`micro2`	Uses timing properties of the MicroSPARC II chip.

Table 2-7 The `-xchip` Values *(Continued)*

	Value	Meaning
Sparc	`hyper`	Uses timing properties of the HyperSPARC™ chip.
	`hyper2`	Uses timing properties of the HyperSPARC II chip.
	`powerup`	Uses timing properties of the Weitek PowerUp™ chip.
	`ultra`	Uses timing properties of the UltraSPARC chip.
PowerPC	`generic`	Uses timing properties for good performance on most PowerPC processors.
	`603`	Uses timing properties of the PowerPC 603 chip.
	`604`	Uses timing properties of the PowerPC 604 chip.
Intel:	`generic`	Uses timing properties for good performance on most Intel x86 processors.
	`386`	Uses timing properties of the Intel 386 chip.
	`486`	Uses timing properties of the Intel 486 chip.
	`pentium`	Uses timing properties of the Intel Pentium chip.
	`pentium_pro`	Uses timing properties of the Intel Pentium Pro chip.

–xF

With `–noex`, enables reordering of functions, using the compiler, the Analyzer, and the linker.

If you compile with the `-xF` option, and then run the Analyzer, you generate a map file that shows an optimized order for the functions. The subsequent link to build the executable file can be directed to use that map by using the linker `-M`*mapfile* option.

If you include the `O` flag in the string of segment flags within the mapfile, then the static linker, `ld`, attempts to place sections in the order in which they appear in the mapfile, for example:

```
LOAD ? RXO
```

Note – The `-xF` option will not work unless you also specify the `–noex` option.

The `analyzer`(1) and `ld`(1) man pages provide further details on this option.

–`xildoff`

Turns off the incremental linker and forces the use of `ld`. This option is the default if you do *not* use the `-g` option. Override this default by using the `-xildon` option.

–`xildon`

Turns on the incremental linker and forces the use of `ild` in incremental mode. This option is the default if you use the `-g` option. Override this default by using the `-xildoff` option.

For more information on the incremental linker, refer to the `ild` man page.

–`xinline=`*rlst*

Same as `-inline`, as described in "–inline=rlst" on page 29.

–`xlibmieee`

Same as `-libmieee`, as described in "–libmieee" on page 31.

–`xlibmil`

Same as `-libmil`, as described in "–libmil" on page 32.

`-xlibmopt`

(SPARC)(Intel) Uses a math routine library optimized for performance. The results may be slightly different from those produced by the normal math library. This option is implied by the `-fast` option.

See also "–fast" on page 22 and "–xnolibmopt" on page 56.

`-xlic_lib=`*l*

(SPARC)*(Intel)* Links in the Sun-supplied, licensed libraries specified in *l*. For example, to link with the Sun Performance Library, use `-xlic_lib=sunperf`. This option, like `-l`, should appear at the end of the command line, after source or object file names.

For further information regarding the performance library, see the `performance_library` README.

`-xlicinfo`

Displays information on the licensing system. In particular, this option returns the license-server name and the user ID for each user who has a license checked out. When you use this option, the compiler is *not* invoked, and a license is *not* checked out.

`-Xm`

Allows the use of the character `$` in identifier names, except as the first character.

`-xM`

Outputs makefile dependency information. For example, with the following code, `hello.c`:

```
#include <stdio.h>
main()
{
    (void) printf ("hello0");
}
```

When you compile with this option:

```
% CC -xM hello.c
```

The output shows the dependencies:

```
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

See `make`(1) for details about makefiles and dependencies.

`–xM1`

Same as `-xM`, except that this option does not output dependencies for the `/usr/include` header files. When you compile the same code provided with the `-xM` option, the output is:

```
hello.o: hello.c
```

`–xMerge`

Merges the data segment with the text segment. The data in the object file is read-only, and is shared between processes, unless you link with `ld -N`.

`–xnolib`

Same as `-nolib`, as described in "–nolib" on page 34.

`–xnolibmopt`

(SPARC)(Intel) Disables `-fast` and does *not* use the math routine library.

Use this option *after* the `-fast` option on the command line, as in:
`CC -fast -xnolibmopt`

`–xO`*level*

Same as `-O`*level*, as described in "–Olevel" on page 36.

`–xpg`

Same as `-pg`, as described in "–pg" on page 39.

`–xprofile=`*p*

Collects data for a profile or uses a profile to optimize.

p must be `collect`, `use`[:*executable name*] or `tcov`. *Executable name* is the name of the executable that is being analyzed. The *executable name* is optional. If it is not specified, the *executable name* is assumed to be `a.out`.

This option causes execution frequency data to be collected and saved during the execution. The data can be used in subsequent runs to improve performance.

The `-xprofile=tcov` and the `-a` options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with `-xprofile=tcov` and others with `-a`. You cannot compile a single file with both options.

Table 2-8 The `-xprofile` Values

Value	Meaning
`collect`	Collects and saves execution frequency for later use by the optimizer. The compiler inserts code to measure the execution frequency at a low level. During execution, the measured frequency data is written into `.prof` files that correspond to each of the source files. If you run the program several times, the execution frequency data accumulates in the `.prof` files; that is, output from prior runs is not lost.
`use[`*:en*`]`	Uses execution frequency data saved by the compiler. Where *en* stands for *executable name*, the name of the executable that is being analyzed. The executable name is optional. If it is not specified, the executable name is assumed to be `a.out`. Optimizes by using the execution frequency data previously generated and saved in the `.prof` files by the compiler. The source files and the compiler options (excepting only this option), must be exactly the same as for the compilation used to create the compiled program that was executed to create the `.prof` files.
`tcov`	This option is the new style of basic block profiling for `tcov`. It has similar functionality to the `-a` option, but correctly collects data for programs that have source code in header files or make use of C++ templates. See `-a` for information on the old style of profiling, the `tcov`(1) man page, and *Profiling Tools* for more details. Code instrumentation is performed similarly to that of the `-a` option, but `.d` files are no longer generated. Instead, a single file is generated, whose name is based on the final executable. For example, if the program is run out of `/foo/bar/myprog`, then the data file is stored in `/foo/bar/myprog.profile/myprog.tcovd`. The `-xprofile=tcov` and the `-a` options are compatible in a single executable. That is, you can link a program that contains some files which have been compiled with `-xprofile=tcov` and others with `-a`. You cannot compile a single file with both options. When running `tcov`, you must pass it the `-x` option to force it to use the new style of data. If you don't, `tcov` uses the old `.d` files by default, and produces unexpected output. Unlike `-a`, the `TCOVDIR` environment variable has no effect at compile time. However, its value is used at program runtime.

-xregs=*r*

(*SPARC*) Specifies the usage of registers for the generated code.

r is a comma-separated list that consists of one or more of the following: `[no%]appl`, `[no%]float`.

Example: `-xregs=appl,no%float`

Table 2-9 The `-xregs` Values

Value	Meaning
`appl`	Allows using the registers `g2`, `g3`, and `g4`. In the SPARC ABI, these registers are described as *application* registers. Using these registers can increase performance because fewer load and store instructions are needed. However, such use can conflict with some old library programs written in assembly code.
`no%appl`	Does not use the `appl` registers.
`float`	Allows using the floating-point registers as specified in the SPARC ABI. You can use these registers even if the program contains no floating-point code.
`no%float`	Does not use the floating-point registers. With this option, a source program cannot contain any floating-point code.

The default is `-xregs=appl,float`.

-xs

Disables Auto-Read for `dbx`. Use this option in case you cannot keep the `.o` files around. This option passes the `-s` option to the assembler.

No Auto-Read is the older way of loading symbol tables. It places all symbol tables for `dbx` in the executable file. The linker links more slowly, and `dbx` initializes more slowly.

Auto-Read is the newer and default way of loading symbol tables. With Auto-Read the information is placed in the `.o` files, so that `dbx` loads the information only if and when it is needed. Hence the linker links faster, and `dbx` initializes faster.

With `-xs`, if you move executables to another directory, when using dbx you do not have to move the object (`.o`) files.

Without `-xs,` if you move the executables to another directory, to use `dbx` you must move *both* the source files and the object (`.o`) files.

`–xsafe=mem`

(*SPARC*) Allows the compiler to assume no memory-based traps occur.

This option grants permission to use the speculative load instruction on V9 machines.

`–xsb`

Same as `-sb`, as described in "–sb" on page 43.

`–xsbfast`

Same as `-sbfast`, as described in "–sbfast" on page 43.

`–xspace`

(SPARC) Does no optimizations that increase code size.

Example: Do not unroll loops.

`–xtarget=`*t*

Specifies the target system for instruction set and optimization.

t must be one of `native`, `generic`, or *system-name*. See Table 2-10 for the meanings of these `–xtarget` values.

The performance of some programs may benefit by providing the compiler with an accurate description of the target computer hardware. When program performance is critical, the proper specification of the target hardware could be very important. This is especially true when running on the newer SPARC processors. However, for most programs and older SPARC processors, the performance gain is negligible and a generic specification is sufficient.

Table 2-10 The `-xtarget` Values

Value	**Meaning**
`native`	Gets the best performance on the host system. The compiler generates code for the best performance on the host system. It determines the available architecture, chip, and cache properties of the machine on which the compiler is running.
`generic`	Gets the best performance for generic architecture, chip, and cache. The compiler expands `-xtarget=generic` to: `-xarch=generic -xchip=generic -xcache=generic` This is the default value.
system-name	Gets the best performance for the specified system. You select a system name from Table 2-11 that lists the mnemonic encodings of the actual system name and numbers.

The `–xtarget` option is a macro. Each specific value for `-xtarget` expands into a specific set of values for the `-xarch`, `-xchip`, and `-xcache` options. See Table 2-9 for the values. For example:

`-xtarget=sun4/15` is equivalent to:
`-xarch=v8a  -xchip=micro  -xcache=2/16/1`

On PowerPC, `–xtarget=` accepts `generic` or `native`.

On Intel, `–xtarget=` accepts:

- `generic` or `native`
- `386` (equivalent to `–386` option) or `486` (equivalent to `–486` option)
- `pentium` (equivalent to `–pentium` option) or `pentium_pro`

Table 2-11 The `-xtarget` Expansions

`-xtarget`	`-xarch`	`-xchip`	`-xcache`
`sun4/15`	`v8a`	`micro`	`2/16/1`
`sun4/20`	`v7`	`old`	`64/16/1`
`sun4/25`	`v7`	`old`	`64/32/1`
`sun4/30`	`v8a`	`micro`	`2/16/1`
`sun4/40`	`v7`	`old`	`64/16/1`
`sun4/50`	`v7`	`old`	`64/32/1`
`sun4/60`	`v7`	`old`	`64/16/1`
`sun4/65`	`v7`	`old`	`64/16/1`
`sun4/75`	`v7`	`old`	`64/32/1`
`sun4/110`	`v7`	`old`	`2/16/1`
`sun4/150`	`v7`	`old`	`2/16/1`
`sun4/260`	`v7`	`old`	`128/16/1`
`sun4/280`	`v7`	`old`	`128/16/1`
`sun4/330`	`v7`	`old`	`128/16/1`
`sun4/370`	`v7`	`old`	`128/16/1`
`sun4/390`	`v7`	`old`	`128/16/1`
`sun4/470`	`v7`	`old`	`128/32/1`
`sun4/490`	`v7`	`old`	`128/32/1`
`sun4/630`	`v7`	`old`	`64/32/1`
`sun4/670`	`v7`	`old`	`64/32/1`
`sun4/690`	`v7`	`old`	`64/32/1`
`sselc`	`v7`	`old`	`64/32/1`
`ssipc`	`v7`	`old`	`64/16/1`
`ssipx`	`v7`	`old`	`64/32/1`
`sslc`	`v8a`	`micro`	`2/16/1`
`sslt`	`v7`	`old`	`64/32/1`

Table 2-11 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/64/1
ssslc	v7	old	64/16/1
ss1	v7	old	64/16/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/64/1
ss5	v8a	micro2	8/64/1
ssvygr	v8a	micro2	8/16/1
ss10	v8	super	16/32/4
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/20	v8	super	16/32/4
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/61	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/402	v8	super	16/32/4
ss10/412	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1

Table 2-11 The `-xtarget` Expansions *(Continued)*

`-xtarget`	`-xarch`	`-xchip`	`-xcache`
`ss10/514`	`v8`	`super`	`16/32/4:1024/32/1`
`ss10/612`	`v8`	`super`	`16/32/4:1024/32/1`
`ss10/712`	`v8`	`super2`	`16/32/4:1024/32/1`
`ss20/hs11`	`v8`	`hyper`	`256/64/1`
`ss20/hs12`	`v8`	`hyper`	`256/64/1`
`ss20/hs14`	`v8`	`hyper`	`256/64/1`
`ss20/hs21`	`v8`	`hyper`	`256/64/1`
`ss20/hs22`	`v8`	`hyper`	`256/64/1`
`ss20/51`	`v8`	`super`	`16/32/4:1024/32/1`
`ss20/61`	`v8`	`super`	`16/32/4:1024/32/1`
`ss20/71`	`v8`	`super2`	`16/32/4:1024/32/1`
`ss20/502`	`v8`	`super`	`16/32/4`
`ss20/514`	`v8`	`super`	`16/32/4:1024/32/1`
`ss20/612`	`v8`	`super`	`16/32/4:1024/32/1`
`ss20/712`	`v8`	`super`	`16/32/4:1024/32/1`
`ss600/41`	`v8`	`super`	`16/32/4:1024/32/1`
`ss600/51`	`v8`	`super`	`16/32/4:1024/32/1`
`ss600/61`	`v8`	`super`	`16/32/4:1024/32/1`
`ss600/120`	`v7`	`old`	`64/32/1`
`ss600/140`	`v7`	`old`	`64/32/1`
`ss600/412`	`v8`	`super`	`16/32/4:1024/32/1`
`ss600/`	`v8`	`super`	`16/32/4:1024/32/1`
`ss1000`	`v8`	`super`	`16/32/4:1024/32/1`
`sc2000`	`v8`	`super`	`16/32/4:1024/64/1`
`cs6400`	`v8`	`super`	`16/32/4:2048/64/1`
`solb5`	`v7`	`old`	`128/32/1`
`solb6`	`v8`	`super`	`16/32/4:1024/32/1`

Table 2-11 The `-xtarget` Expansions *(Continued)*

-xtarget	-xarch	-xchip	-xcache
ultra	v8	ultra	16/32/1:512/64/1
ultra1/140	v8	ultra	16/32/1:512/64/1
ultra1/170	v8	ultra	16/32/1:512/64/1
ultra1/1170	v8	ultra	16/32/1:512/64/1
ultra1/2170	v8	ultra	16/32/1:512/64/1
ultra1/2200	v8	ultra	16/32/1:1024/64/1

–xtime

Same as `-time`, as described in "–time" on page 45.

–xunroll=*n*

Same as `-unroll=`*n*, as described in "–unroll=n" on page 45.

–xwe

Converts all warnings to errors by returning non zero exit status.

–Ztha

(SPARC) Prepares code for analysis by the Thread Analyzer, the performance analysis tool for multithreaded code.

If you compile and link in separate steps, you should use this option in both steps. Code linked with this option is linked with the `libtha.s` library.

–ztext

Forces a fatal error if any relocations remain against non writable, allocatable sections.

Templates 3

This chapter discusses both template use and compilation. It introduces template concepts and terminology in the context of function templates, discusses the more complicated (and more powerful) class templates, and the nested use of templates. Also discussed are the emerging standard for template specialization and file organization in the presence of templates. Other topics include: the template compilation process, instance linkage options, automatic instance consistency, and searching for template definitions.

The purpose of templates is to enable programmers to write a single body of code that applies to a wide range of types in a type-safe manner. Before reading this chapter, you should be familiar with the discussion of templates in *C++ Annotated Reference Manual.*

The C++ compiler's implementation of templates is based on the *C++ Annotated Reference Manual* with elements from the emerging ISO C++ standard.

The C++ compiler's implementation of templates is different from that of AT&T's `Cfront` compiler. See "Compile-Time Versus Link-Time Instantiation" on page 84. For additional information, refer to Appendix A, "Migration Guide".

Function Templates

A function template describes a set of related functions. That set is parameterized by types.

Template Declaration

Templates must be declared before they can be used. 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 );
```

`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.

Template Definition

If a template is declared, it must also be defined. 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 may be repeated several times. All definitions, however, must be the same. This restriction is called the One-Definition Rule.

Template Use

Once declared, templates may be used like any other function. Their *use* consists of naming the template and providing *template arguments* corresponding to the template's parameters. For example, you may use the previously declared template as follows, explicitly specifying the template argument.

```
double example1( double item )
    { return twice<double>( item ); }
```

In this case, `double` is the type argument. Often the compiler will be able to infer the template arguments from the function arguments. When this is the case, you may omit argument specifications, as in:

```
double example2( double item )
    { return twice( item ); }
```

If the compiler is unable to uniquely infer template arguments from the function arguments, it will issue an error. To correct the problem, simply specify the template arguments.

Template Instantiation

The process of *instantiation* involves generating a concrete function (instance) for a particular combination of template arguments. For example, the compiler will generate a function for `twice<int>` and a different function for `twice<double>`. Instantiation may be implicit or explicit.

Implicit Instantiation

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

Explicit Instantiation

The compiler will implicitly instantiate 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 in the following example:

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

Template parameters and arguments may be omitted when the compiler can infer them, as in:

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

Class Templates

A class template describes a set of related classes, or data types. That set is parameterized by types, by integral values, and/or by some variable references. Class templates are particularly useful in describing generic, but type-safe, data structures.

Template Declaration

A class template declaration must declare the class data and function members, as in the examples below. In these examples, template arguments are `Size`, an `unsigned int`, and `Elem`, a type.

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

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

Template Definition

The definition of a class template consists of a template definition for its member functions, and for its static data members. Dynamic (non-static) data members are sufficiently defined by the template declaration.

Function Members

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

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

```
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 *inital )
    { strncpy( data, initial, Size );
      if ( length( ) == Size ) overflow = 1; }
```

Static Data Members

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

Static data member definitions must always provide an initial value, otherwise, the statement is interpreted as a declaration.

Template Use

A template class may be used wherever a type may be used. Specifying a template class consists of providing the template name and the template arguments corresponding to the template parameters. The following examples define two variables, one of array and one string.

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

```
String<8> short_string( “hello” );
```

You may use template class member functions as you would any other member function. For example:

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

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

Instantiation

The process of instantiation involves generating a concrete class (instance) for a particular combination of template arguments. For example, the compiler will generate 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 “Template Declaration”, “Template Definition”, and “Template Use” sections, the compiler substitutes `int` wherever `Elem` appears.

When instantiating a template class, the compiler must instantiate the function members and the static data members. The dynamic data members are not part of the instantiation; they are part of the variable declared with the template class as its type.

Implicit Instantiation

The use of a template class introduces the need for an instance. If that instance does not already exist, the compiler will generate it.

Explicit Instantiation

Implicit instantiation may be inappropriate when constructing libraries that provide templates. Templates may be explicitly instantiated, as in:

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

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

When the programmer explicitly instantiates a class, all of its member functions are also instantiated.

Whole Class Instantiation

When the compiler implicitly instantiates a template class, it instantiates the static data members, the constructor, and the destructor. However, the compiler does not implicitly instantiate any other member function unless the function is explicitly referenced. 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 `–template=no%wholeclass`, which is the default.

Nested Template Use

Templates may be used (though not defined) in a nested manner. This is particularly useful in defining generic functions over generic data structures, as is done in the emerging Standard Template 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; } }
```

This example defines a sort function over the predeclared `Array` class template objects. The following examples shows the actual use of the sort function.

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

Template Specialization, Standard Method

There may be significant performance advantages to treating some combinations of template arguments as a special case, as in the examples below for `twice`. Alternatively, a template description may fail to work for a set of its possible arguments, as in the examples below for `sort`. Template specialization enables the definition of alternate implementations for a given combination of actual template arguments. The template specialization overrides the default instantiation.

Declaration

A specialization must be declared 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 );
```

The template arguments may be omitted if the compiler can unambiguously determine them. For example:

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

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

Definition

All template specializations declared must also be defined. The following examples define the functions declared above.

```
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; } }
```

Use and Instantiation

Specializations are used and instantiated just like any other template, except that the definition of a completely specialized template is also an instantiation.

Template File Organization

There are two primary organizations of template files, the definitions-included and the definitions-separated. The definitions-included organization enables more control over template compilation, and is recommended for that reason.

Definitions-Included

When the declarations and definitions for a template are contained within the file that uses the template, the file 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 the file that uses a template includes a file containing both the template declaration and the template definition, the file 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<int>( -3 ); }
```

It is very important to make template headers *idempotent*. That is, when a header is included many times, its effect is always the same. This is easiest to accomplish when the header turns itself off with `#define` and `#ifndef`, as in the preceding example.

Definitions-Separate

When the file that uses a template includes a file containing only the declaration of the template, and not the definition, the template file has the definitions-separate organization. The definition must appear in another file. For example:

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

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

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

Because of the separation of header and template source files, you must be very careful in file construction, placement, and naming. You may also need to explicitly identify the location of the source file to the compiler.

Potential Problem Areas

This section describes potential problem areas in using templates.

Non-Local Name Resolution and Instantiation

Some names used within a template definition may not be 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. These alternatives may yield different resolutions. Name resolution is complex, and currently under debate in the C++ standards committee. Consequently, you should not rely on non-local names, except those provided in a pervasive global environment. In other words, a name may have different meanings in different places; use only non-local names that are declared everywhere and that mean the same thing everywhere. For example:

`use1.cc`	
	```
	typedef int intermediary;
	int temporary;
	template <class Source, class Target>
	Target converter( Source source )
	       { temporary = (intermediary)source;
	       return (Target)temporary; }
	```

`use2.cc`	
	```
	typedef double intermediary;
	unsigned int temporary;
	template <class Source, class Target>
	Target converter( Source source )
	       { temporary = (intermediary)source;
	       return (Target)temporary; }
	```

In this example, the template function converter uses the non-local names `intermediary` and `temporary`. These names have different definitions in the environment of each template, and will probably yield different results under different compilers. In order for templates to work reliably, all non-local names (`intermediary` and `temporary`) must have the same definition everywhere.

A common use of non-local 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`

```
#include "array.h"
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

The template instantiation system requires that a declaration follow the search rules in locating template definitions, that is, a declaration must precede an instantiation. This declaration must be a true declaration, and not a `friend` declaration. For example:

`array.h`	
	```
	#ifndef _ARRAY_H
	#define _ARRAY_H
	#include <iostream.h>
	template <class T> class array {
	    protected:
	        int size;
	    public:
	        array( );
	        friend ostream& operator<<( ostream&, const array<T>& );
	};
	#endif // _ARRAY_H
	```

`array.cc`	
	```
	#include <iostream.h>
	#include <stdlib.h>
	#include "array.h"
	template <class T> array<T>::array( )
	    { size = 1024; }
	template <class T>
	ostream& operator<<( ostream& out, const array<T>& rhs )
	    { return out << "[" << rhs.size << "]"; }
	```

`main.cc`	
	```
	#include <iostream.h>
	#include "array.h"
	int main( )
	    { cout << "creating an array of int... " << flush;
	      array<int> foo;
	      cout << "done\n";
	      cout << foo << endl;
	      return 0; }
	```

When the compilation system attempts to link the produced object files, it will generate an undefined error for the `operator<<` function, which is *not* instantiated. There is no error message during compilation because the compiler will have read:

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

as the declaration of a normal function that is a `friend` of the `array` class. To properly instantiate the functions, the following declaration must be added outside the array class declaration:

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

This declaration guarantees that the function is instantiated for each object of type `array<T>`.

Template Compilation

Template compilation is a complicated process because the compiler must instantiate templates at nearly arbitrary times during compilation, and because it may need to search for template definitions among several files. This section describes the major components of template compilation.

Verbose Compilation

The C++ compiler will notify the users of significant events during template compilation when given the flag `-verbose=template`. Conversely, the compiler will not notify users when given `-verbose=no%template`, which is the default. The `+w` option may give other indications of potential problems when template instantiation occurs.

Template Database

The template database is a directory containing all configuration files needed to handle and instantiate the templates required by your program. It also acts as a repository for all generated object files containing templates.

The purpose of the template database is to ensure that there is exactly one instance for every combination of template and arguments; that is, to ensure that there are no redundant template instances. The database creates instances only when necessary, ensuring that all instances are up-to-date.

The template database is only used during external instantiation (see "External Instance Linkage" on page 87).

The template database is contained, by default, in the subdirectory `Templates.DB` within the current directory. You may specify the directory with the `-ptr`*directory* option. If the directory does not exist, and the compiler needs to instantiate a template, the directory is created for you.

Multiple Template Databases

You may specify multiple databases by giving multiple `-ptr` options. When specifying multiple databases, the first one to specify is your working, writable database. All others are read-only. To avoid confusion in multiple-database environments, make the first entry your current working directory. For example:

```
CC -ptr. -ptr/usr/lib/Templates.DB -o main main.cc
```

Note – It is highly recommended that you do not rely on the use of the `-ptr` directive as its use may become obsolete in future releases. While alternatives will be available, the direct use of `-ptr` might cause compiler performance degradation in the future.

Using the Same Template Database for Multiple Targets

If you use the same template database for multiple targets, inconsistent results can occur. However, if you choose to build multiple targets in the same work directory, `tdb_link` notes this fact during parallel builds and implements a locking mechanism on the primary working database. If two or more links are attempted on the same database, `tdb_link` initiates a wait until the lock is cleared by the other compilation.

The `ptclean` *Command*

Changes in your program can render some instantiations superfluous, thus wasting storage space. The `ptclean` command clears out the template database, removing all instantiations, temporary files, and dependency files. Instantiations will be recreated when, and only when, needed.

Options File

The template options file, `Template.opt`, is a user-provided optional file that contains options needed to locate template definitions and to control instance recompilation. In addition, the options file provides features for controlling template specialization and explicit instantiation, although the user is discouraged from using them because the C++ compiler now supports the syntax required to declare specializations and explicit instantiation in the source code.

The options file is an ASCII text file containing a number of entries. An entry consists of a keyword followed by expected text and terminated with a semicolon (;). Entries can span multiple lines, although the keywords cannot be split.

Comment Entries

Comments start with a # character and extend to the end of the line. Text within a comment is ignored.

```
# Comment text is ignored until the end of the line.
```

Sharing Options Files

You may share options files among several template databases by including the options files. This facility is particularly useful when building libraries containing templates. During processing, the specified options file is textually included in the current options file. You can have more than one `include` statement and place them anywhere in the options file. The options files can also be nested.

```
include "options-file";
```

Template Definition Location and Consistency Entries

The `extensions` and `definition` options file entries are described in “Template Definition Searching” on page 87.

Template Specialization Entries

The `special` options file entries are described in “Template Specialization, Deprecated Method” on page 92.

Compile-Time Versus Link-Time Instantiation

Instantiation is the process by which a C++ compiler creates a usable function or object from a template. Two common methods of template instantiation are compile-time instantiation and link-time instantiation.

C++ Compile-Time Instantiation

C++ 4.2 uses compile-time instantiation, which forces instantiations to occur when the reference to the template is being compiled.

The advantages of compile-time instantiation are:

- Debugging is much easier—error messages occur within context, allowing the compiler to give a complete traceback to the point of reference.
- Template instantiations are always up to date.
- The overall compilation time, including the link phase, is reduced.

Templates may be instantiated multiple times if source files reside in different directories, or if you use libraries with template symbols.

`Cfront` *Link-Time Instantiation*

`Cfront` uses the link-time method, which uses this algorithm:

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

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

The two main disadvantages of link-time instantiation are:

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

Template Instance Linkage

You may instruct the compiler to use one of three different instance linkage methods: static, global, or external. Static instances are suitable for very small programs or debugging. Global instances are suitable for some library construction. External instances are suitable for all development and provide the best overall template compilation. External instances are the default.

Note – If you wish to create a library that contains all instances of the templates that it uses, specify the `–xar` option when creating the library. Do *not* use the `ar` command. For example: **`CC –xar libmain.a a.o b.o c.o`**

Static Instance Linkage

Under static instance linkage, all instances are placed within the current compilation unit. As a consequence, templates will be re-instantiated during each re-compilation; instances are not saved to the template database.

Template instances receive static linkage. These instances will not be visible or usable outside of the current compilation unit.

Specify static instance linkage with the `-instances=static` option.

Static instance linkage may only be used with the definitions-included template organization. The compiler will not search for definitions.

Templates may have identical instantations in several object files. This results in unnecessarily large programs. Static instance linkage is therefore suitable only for small programs, where templates are unlikely to be multiply instantiated.

Compilation is potentially faster with static instance linkage, so static linkage may also be suitable during fix-and-continue debugging.

Global Instance Linkage

Under global instance linkage, all instances are placed within the current compilation unit. As a consequence, templates will be re-instantiated during each re-compilation; they are not saved to the template database.

Template instances receive global linkage. These instances will be visible and usable outside of the current compilation unit.

Specify global instance linkage with the `-instances=global` option.

Global instance linkage may only be used with the definitions-included template organization. The compiler will not search for definitions.

Templates may have identical instantations in several object files. This results in multiple definitions conflicts during linking. Global linkage is therefore suitable only when it is known that instances will not be repeated, such as when constructing libraries with explicit instantiation.

External Instance Linkage

Under external instance linkage, all instances are placed within the template database. Templates will be re-instantiated only when necessary.

Instances are referenced from the current compilation unit with external linkage. Instances are defined within the template database with global linkage. The compiler ensures that exactly one template instance exists; instances will be neither undefined nor multiply defined.

Specify external linkage with the `-instances=extern` option. This option is the default.

Because instances are stored within the template database, you must use the `CC` command to link C++ objects that use external instance linkage into programs.

If you wish to create a library that contains all instances that it uses, specify the `–xar` option when creating the library. Do *not* use the `ar` command. For example:

```
CC –xar libmain.a a.o b.o c.o
```

Template Definition Searching

When using the definitions-separate template file organization, template definitions are not available in the current compilation unit, and the compiler must search for the definition. This section describes the controls on that search. Definition searching is somewhat complex and prone to error. Therefore, you should use the definitions-included template file organization, which avoids definition searching altogether.

Source File Location Conventions

Without specific directions as provided with an options file (see below), the compiler uses a `Cfront`-style method to locate template definition files. This method requires that the template definition file contain the same base name as the template declaration file, and that it also be on the current `include` path. For example, if the template function `foo()` is located in `foo.h`, the matching template definition file should be named `foo.cc` or some other recognizable source-file extension. The template definition file must be located in one of the normal `include` directories or in the same directory as its matching header file.

Source File Extensions

You can specify different source file extensions for the compiler to search for when it is using its default `Cfront`-style source-file-locator mechanism. The format is:

```
extensions "ext-list";
```

The *ext-list* is a list of extensions for valid source files in a space-separated format such as:

```
extensions ".CC .c .cc .cpp";
```

In the absence of this entry from the options file, the valid extensions for which the compiler searches are `.cc`, `.c`, `.cpp`, `.C`, and `.cxx`. (See “Options File” on page 83.)

Definitions Search Path

As an alternative to the normal search path set with `–I`, you may specify a search directory for template definition files with the option `-pti`*directory*. Multiple `-pti` flags will define multiple search directories, i.e. a search path. If `-pti`*directory* is used, the compiler looks for template definition files on this path and ignores the `-I` flag. It is recommended, however, that you use the `-I` flag instead of the `-pti`*directory* flag, since the `-pti`*directory* flag complicates the search rules for source files.

Options File Definition Entries

Definition source file locations may be explicitly specified with the `definition` option file entry. The definition entry is provided for those cases when the template declaration and definition file names do not follow the standard `Cfront`-style conventions. See “Source File Location Conventions” on page 88. The entry syntax is:

```
definition name in "file-1",[ "file-2" ..., "file-n"] [nocheck "options"];
```

The *name* field indicates the template for which the option entry is valid. Only *one* definition entry per name is allowed. That name must be a simple name; qualified names are *not* allowed. Parentheses, return types and parameter lists are not allowed. Regardless of the return type or parameters, only the name itself counts. As a consequence, a definition entry may apply to several (possibly overloaded) templates.

The "*file-n*" list field specifies in which files the template definitions may be found. Search for the files using the definition search path described in the previous section, “Definitions Search Path.”The file names *must* be enclosed in quotes (" "). Multiple files are available because the simple template name may refer to different templates defined in different files, or because a single template may have definitions in multiple files. For example, if `func` is defined in three files, then those three files *must* be listed in the definition entry.

The `nocheck` field is described in “Template Instance Automatic Consistency” on page 91.

In the following example, the compiler locates the template function `foo` in `foo.cc`, and instantiates it. In this case, the definition entry is redundant with the default search.

`foo.cc`	`template <class T> T foo( T t ) { }`
`Template.opt`	`definition foo in “foo.cc”;`

The following example shows the definition of static data members and the use of simple names.

`foo.h`	`template <class T> class foo { static T* fooref; };`

`foo_statics.cc`	`#include “foo.h”`
	`template <class T> T* foo<T>::fooref = 0`

`Template.opt`	`definition fooref in “foo_statics.cc”;`

The name provided for the definition of `fooref` is a simple name and not a qualified name (such as `foo::fooref`). The reason for the definition entry is that the file name is not `foo.cc` (or some other recognizable extension) and cannot be located using the default `Cfront`-style search rules. See “Source File Location Conventions” on page 88.

The following example shows the definition of a template member function. As the example shows, member functions are handled exactly like static member initializers.

`foo.h`	`template <class T> class foo { T* foofunc(T); };`

`foo_funcs.cc`	`#include “foo.h”`
	`template <class T> T* foo<T>::foofunc(T t) {}`

`Template.opt`	`definition foofunc in “foo_funcs.cc”;`

The following example shows the definition of template functions in two different source files.

`foo.h`	`template <class T> class foo {`
	`    T* func( T t );`
	`    T* func( T t, T x );`
	`};`

`foo1.cc`	`#include "foo.h"`
	`template <class T> T* foo<T>::func( T t ) { }`

`foo2.cc`	`#include "foo.h"`
	`template <class T> T* foo<T>::func( T t, T x ) { }`

`Template.opt`	`definition func in "foo1.cc", "foo2.cc";`

In this example, the compiler must be able to find both definitions of the overloaded function `func()`. The definition entry tells the compiler where to find the appropriate function definitions.

Template Instance Automatic Consistency

The template database manager ensures that the state of the files in the database is consistent and up-to-date with your source files.

For example, if your source files are compiled with `-g` (debugging on), the files you need from the database are also compiled with `-g`.

In addition, the template database tracks changes in your compilation. For example, if the first time you compile your sources, you have the `-DDEBUG` flag set to define the name `DEBUG`, the database tracks this. If you omit this flag on a subsequent compile, the compiler reinstantiates those templates on which this dependency is set.

Options File Nocheck Field

Sometimes recompiling is unnecessary when certain compilation flags change. You can avoid unnecessary recompilation using the `nocheck` field of the `definition` option file entry, which tells the compiler and template database manager to ignore certain options when checking dependencies. If you do not want the compiler to reinstantiate a template function because of the addition or deletion of a specific command-line flag, you should add that flag here. The entry syntax is:

```
definition name in "file-1"[, "file-2" ..., "file-n"] [nocheck "options"];
```

The name and file list fields are described in "Template Definition Searching" on page 87. The options themselves must be enclosed in quotes (" ").

In the following example, the compiler locates the template function `foo` in `foo.cc`, and instantiates it. If a reinstantiation check were later required, the compiler would ignore the `-g` option.

`foo.cc`	`template <class T> T foo( T t ) {}`
`Template.opt`	`definition foo in "foo.cc" nocheck "-g";`

Template Specialization, Deprecated Method

Until recently, the C++ language provided no mechanism for specializing templates, so each compiler provided its own mechanism. This section describes the specialization of templates using the mechanism of previous versions of the C++ compilers.

Note – Since this mechanism is supported in this release of the C++ compiler, but may not be supported in future releases, it is referred to as "deprecated." Avoid this mechanism for new code, and migrate existing code to the standard mechanism. (See "Template Specialization, Standard Method" on page 74.)

Specialized Template Definitions

The following example shows a specialized constructor that creates an array of character strings of fixed size:

```
void Array<char*>::Array( int sz )
    { size = sz;
      data = new char * [size];
      for ( int i = 0; i < size; i++ ) data[i] = new char[128]; }
```

In this example, the constructor allocates each `char` pointer in the array. Because a given executable may span many object files and libraries, the compiler does not know whether the specific function called exists elsewhere in the compilation environment. Specializations can be registered in the options file, as discussed below. Given the preceding code, the following takes place:

```
Array<int>   IntArray( 10 );      // The compiler instantiates an
                               // Array of ints using the default
                                   // constructor
Array<char*> CharStarArray( 10 ); // The compiler uses the
                                   // specialized constructor
```

The following example shows specialization of template functions:

```
template <class T> void func ( T t ) { }    // A template function
void func( double ) { }                  // A specialization of func()
int main( )
    { func(10);    // The compiler-instantiated func<int> is used
      func(1.23);  // The user-provided func(double) is used
    }
```

Options File Specialization Entries

The `special` entry specifies to the compiler that a given function is a specialization and should not be instantiated when encountered. When using the compile-time instantiation method, preregister specializations with an entry in the options file. The syntax is:

`special` *declaration*;

The declaration is a legal C++-style declaration without return types. Overloading of the `special` entry is allowed. For example:

`foo.h:`	`template <class T> T foo( T t ) { };`
`main.cc:`	`#include “foo.h”`
`Template.opt:`	`special foo(int);`

The preceding options file informs the compiler that the template function `foo()` should not be instantiated for the type `int`, and that a specialized version is provided by the user. Without that entry in the options file, the function may be instantiated unnecessarily, resulting in errors:

`foo.h`	`template <classT> T foo( T t ) { return t + t; }`

`file.cc`	`#include "foo.h"`
	`int func( ) { return foo( 10 ); }`

`main.cc`	`#include "foo.h"`
	`int foo( int i ) { return i * i; } // the specialization`
	`int main( ) { int x = foo( 10 ); int y = func(); return 0; }`

In the preceding example, when the compiler compiles `main.cc`, the specialized version of `foo` is correctly used because the compiler has seen its definition. When `file.cc` is compiled, however, the compiler instantiates its own version of `foo` because it doesn't know `foo` exists in `main.cc`. In most cases, this process results in a multiply-defined symbol during the link, but in some cases (especially libraries), the wrong function may be used, resulting in runtime errors. If you use specialized versions of a function, you *should* register those specializations.

The `special` entries can be overloaded, as in this example:

`foo.h`	`template <classT> T foo( T t ) {}`

`main.cc`	`#include "foo.h"`
	`int   foo( int   i ) {}`
	`char* foo( char* p ) {}`

`Template.opt`	`special foo(int);`
	`special foo(char*);`

To specialize a template class, include the template arguments in the `special` entry:

`foo.h`	`template <class T> class Foo { ... various members ... };`

`main.cc`	`#include "foo.h"`
	`int main( ) { Foo<int> bar; return 0; }`

`Template.opt`	`special class Foo<int>;`

If a template class member is a static member, you must include the keyword `static` in your specialization entry:

`foo.h`	`template <class T> class Foo { public: static T func(T); };`

`main.cc`	`#include "foo.h"`
	`int main( ) { Foo<int> bar; return 0; }`

`Template.opt`	`special static Foo<int>::func(int);`

Exception Handling 4

This chapter explains exception handling as currently implemented in the C++ compiler. It also identifies some areas of exception handling that are not clearly defined in the ANSI X3J16 draft working paper. Some of the topics covered in this chapter are interpretations of the draft. Other topics are specific to this compiler implementation.

For additional information on exception handling, see *The C++ Programming Language (*Second Edition*)* by Margaret A. Ellis and Bjarne Stroustrup.

Why 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.

4

Using Exception Handling

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 may cause an exception. This grouping restricts exception handlers to exceptions generated within the `try` block.

`catch`

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

- The keyword `catch`
- A `catch` expression, 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 to a subsequent exception handler. A `throw` statement is specified with:

- The keyword `throw`
- An assignment expression; the type of the result of the expression determines which `catch` exception handler receives control

An Example

Code Example 4-1 Exception Handling Example

```
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
    }
}
```

In this 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.

Implementing Exception Handlers

Here are the basic tasks involved in implementing exception handlers:

- When a function is called by many other functions, you should code it so 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. Precede function calls that you anticipate may produce an exception with the keyword `try` and enclose the calls in braces.

- 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 `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 outermost `catch` block.
  - 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 current draft supports synchronous exception handling with 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, exception handling can be made to work in the presence of asynchronous events if care is taken. For instance, to make exception handling work with signals, you can write a signal handler that sets a global variable, have another routine that polls the value of that variable at regular intervals, and throws an exception when the value changes.

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 function 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 either rethrows the exception, branches to a label, or ends normally. If a `catch` block ends normally, without a `throw`, the flow of control passes over all subsequent `catch` blocks.

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 that function are safely destroyed via calls to their destructors.

If a `try` block ends without an exception, all subsequent `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 occurs when exceptions are thrown in a handler. The handled exception needs to be kept around so that it can be rethrown. Nesting also occurs when exceptions are thrown in the destructor during stack unwinding.

Using `throw` in a Function Declaration

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

This declaration indicates to the caller that the function `foo` generates only one exception, and that it is caught by a handler of type `X`:

```
void foo(int) throw(X);
```

A variation on the previous example is:

```
void foo(int) throw();
```

This declaration guarantees that no exception is generated by the function `foo`. If an exception occurs, it results in a call to the predefined function `unexpected()`. By default, `unexpected()` calls `abort()` to exit the program. This default behavior can be changed by calling the `set_unexpected()` function; see "set_terminate() and set_unexpected() Functions" on page 103.

The check for unexpected exceptions is done at program execution time, not at compile time. The compiler may, 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.

Runtime Errors

There are five runtime error messages associated with exceptions:

1. `No handler for the exception.`

```
2. Unexpected exception thrown.

3. An exception can only be re-thrown in a handler.

4. During stack unwinding, a destructor must handle its own
   exception.

5. Out of memory.
```

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

Caution – Selecting a `terminate()` function that does not terminate is an error.

The class `xunexpected` is now defined in `exception.h`.

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

`set_terminate()` *and* `set_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()`:

```
typedef void (*PFV)();
PFV set_terminate(PFV);
```

terminate() calls the function passed as an argument to set_terminate(). 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().

set_unexpected()

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

```
typedef void (*PFV)()
PFV set_unexpected(PFV);
```

unexpected() calls the function passed as an argument to set_unexpected(). 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().

Matching Exceptions With Handlers

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

1. T is same as E
2. T is const or volatile of E
3. E is const or volatile of T
4. T is ref of E or E is ref of T
5. T is a public base of E
6. T and E are both pointer types, and E can be converted to T by standard pointer conversion.

Throwing exceptions of reference or pointer types can result in a dangling pointer.

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 and possibly truncates the object, which can happen when the exception is derived from X.

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 is invoked.

Access Control in Exceptions

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

1. The formal argument of a catch clause obeys the same rules as an argument of the function in which the catch clause occurs.
2. 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 104.

-noex Compiler Option

If you know that exceptions are not used, use the compiler option -noex to suppress generation of code that supports exception handling. The use of -noex results in smaller code size and faster code execution. When files compiled with -noex are linked to files compiled without -noex, some local objects are not destroyed when exceptions occur. By default, the compiler generates code to support exception handling.

New Runtime Function and Predefined Exceptions

You can use several functions related to exception handling. The header file exception.h includes the predefined exceptions xmsg and xalloc. The definitions provided in exception.h differ from those in the X3J16 Draft Working Paper.

Default `new-handler()` *Function*

When `::operator new()` cannot allocate storage, it calls the currently installed *new-handler* function. The default *new-handler* function throws an `xalloc` exception.

Note – The old behavior was to return a null from `::operator new()` when a memory request could not be satisfied. To restore the old behavior, call `set_new_handler(0)`.

Building Shared Libraries With Exceptions

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

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 “set_terminate() and set_unexpected() Functions” on page 103.

Runtime Type Information 5

This chapter explains the use of Runtime Type Information (RTTI).

In C++, pointers to classes have a *static* type, which is 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.

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

Here, `ap` has the static type A* and a dynamic type B*.

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

RTTI Options

RTTI support requires significant resources to implement. To enable RTTI implementation and to enable recognition of the associated `typeid` keyword, use the option `-features=rtti`. To disable RTTI implementation and to disable recognition of the associated `typeid` keyword, use the option `-features=no%rtti` (default).

typeid *Operator*

The typeid operator produces a reference to an object of class type_info, which describes the most-derived type of the object. In order to make use of the typeid() function, source code must #include the <typeinfo.h> header file. The primary value of this operator/class combination is in comparisons. In such comparisons, the top-level *const-volatile* qualifiers are ignored, as in the following example.

```
#include <typeinfo.h>
#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 will raise 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, multi-byte string, suitable for conversion and display.

The constructor is a private member function, so there is no way for a programmer to create a variable of type "type_info". The only source of "type_info" objects is in the "typeid" operator.

Cast Operations 6

This chapter explains the new cast operations: const and volatile casts, reinterpret cast, static and dynamic casts.

The emerging C++ standard defines new cast operations that provide finer control over casting than previous cast operations. The `dynamic_cast<>` operator provides a way to check the actual type of a pointer to a polymorphic class. 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 the programmer's intent more clearly and allow the compiler to better check the code.

Cast Operations Options

To enable recognition of the cast operators, use the option `-features=castop`, which is the default. To disable recognition of the cast operators, use the option `-features=no%castop`.

Const and Volatile Cast

The expression `const_cast<T>(v)` can be used to change the "const" or "volatile" qualifiers of pointers or references. `T` must be a pointer, reference, or pointer to member type. If *cv1* and *cv2* are some combination of `const` and `volatile` qualifiers (that is, *cv1* is `volatile` and *cv2* is `const volatile`), `const_cast` can convert a value of type "pointer to *cv1* `T`" to "pointer to *cv2*

T”, or “pointer to member of type *cv1* T” to “pointer to member of type *cv2* T”. If we have an lvalue of type *cv1* T, then `const_cast` can convert it to “reference to type *cv2* T”. (An lvalue names an object in such a way that its address can be taken.)

```
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 will convert from pointer types to integers and back again, between two unrelated pointers, pointers to members, or pointers to functions. The only guarantee on such casts is that a cast to a new type, followed by a cast back to the original type, will have the original value. It is legal to cast an lvalue of type `T1` to type `T2&` if a pointer of type T1* can be converted to a pointer of type `T2*` by a `reinterpret_cast`. `reinterpret_cast` cannot be used to convert between pointers to two different classes that are related by inheritance (use `static_cast` or `dynamic_cast`), nor can it be used to cast away `const` (use `const_cast`).

```
class A { public: virtual void f( ); };
void use_of_reinterpret_cast( )
    { A a1;
      const A a2;
      int i = reinterpret_cast<int>(&a1);   // grab address
     const int j = reinterpret_cast<int>(&a2); }  // grab address
```

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 cast that is performed implicitly on assignment. In addition, any value may be cast to `void`, and any implicit cast can be reversed if that cast would be legal as an old-style cast. It cannot be used to cast away `const`.

```
class B           { public: virtual void g( ); };
class C : public B { public: virtual void g( ); };

void use_of_static_cast( )
    { C c;
      // an explicit temporary lvalue to the base of c, a B
      B& br = c;
      br.g( );   // call B::g instead of C::g
      // a static_cast of an lvalue to the base of c, a B
      static_cast<B&>(c).g( ); }   // call B::g instead of C::g
```

Dynamic Cast

A pointer or reference to a class can actually point to any class publicly derived from that class. Occasionally, it may be desirable to obtain a pointer to the fully-derived class, or to some other base class for the object. The dynamic cast provides this facility.

The dynamic type cast will convert a pointer or reference to one class into a pointer or reference to another class. That second class must be the fully-derived class of the object, or a base class of the fully-derived class.

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, or “pointer to *cv* `void`”, where *cv* is [`const`][`volatile`]. In the case of pointer types, if the specified class is not a base of the fully derived class, the cast returns a null pointer. In the case of reference types, if the specified class is not a base of the fully derived class, the cast throws a `bad_cast` exception. For example, given the class definitions:

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

The following function will succeed.

```
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 will succeed:

```
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 );
     }
```

If run-time type information has been disabled, i.e. `-features=no%rtti`, (See Chapter 5, “RTTI”), the compiler converts `dynamic_cast` to `static_cast` and issues a warning.

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

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 et al.

Moving From C to C++ 7

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

C programs generally require little modification to compile as C++ programs. C and C++ are link compatible. You don't have to modify compiled C code to link it with C++ code.

See *The C++ Programming Language*, by Margaret A. Ellis and Bjarne Stroustrup, for more specific information on the C++ language.

Reserved and Predefined Words

Table 7-1 shows all reserved keywords in C++ and C, plus keywords that are predefined by C++. Keywords that are reserved in C++ and not in C are shown in **`boldface.`**

Table 7-1 Reserved Keywords

asm	do	if	return	typedef
auto	double	**inline**	short	**typeid**
bool	**dynamic_cast**	int	signed	union
break	else	long	sizeof	unsigned
case	enum	**mutable**	static	**using**
catch	**explicit**	**namespace**	**static_cast**	**virtual**
char	**export**	**new**	struct	void
class	extern	**operator**	switch	volatile
const	**false**	**private**	**template**	**wchar_t**
const_cast	float	**protected**	**this**	while
continue	for	**public**	**throw**	
default	**friend**	register	**true**	
delete	goto	**reinterpret_cast**	**try**	

`__STDC__` is predefined to the value 0. For example:

Code Example 7-1 Predefines

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

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

produces:

```
yes
yes
```

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

Table 7-2 Reserved Words for Operators and Punctuators

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

Data Types

The basic (Solaris) C and C++ datatypes and their sizes are: (The size of each numeric type may vary among vendors.)

- `char` (1 byte)
- `short int` (2 bytes)
- `int` (4 bytes)
- `long int` (4 bytes)
- `long long int` (8 bytes)
- `wchar_t` (4 bytes)
- `enum` (4 bytes)

Each of `char`, `short`, `int`, `long`, and `long long` can be prefixed with `signed` or `unsigned`. A type specified with `signed` is the same as the type specified without `signed`, except for `char` and bitfields.

- `float` (4 bytes)
- `double` (8 bytes)
- `long double` (12 bytes on Intel, 16 otherwise)
- `void`

Creating Generic Header Files

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

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

```
#ifdef __cplusplus
int printf(char*,...);// C++ declaration
#else
int printf();/* C declaration */
#endif
```

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

Linking to C Functions

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

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

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

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

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

Fortran 77 Interface 8

Introduction

This chapter describes the Fortran 77 interface with C++. We suggest you follow these steps:

1. Study Code Example 8-1 and “Sample Interface” on page 122.
2. Read “Fortran Calls C++” on page 129 or “C++ Calls Fortran” on page 153.
3. Within either of the two sections mentioned in step 2, choose one of these subsections:
   - Arguments Passed by Reference
   - Arguments Passed by Value
   - Function Return Values
   - Labeled Common
   - Sharing I/O
   - Alternate Returns
4. Within any of the subsections mentioned in step 3, choose one of these examples:

   For the arguments, there is an example for each of these:
   - Simple Types (`character*1`, `logical`, `integer`, `real`, `double precision`)
   - Complex Types (`complex`, `double complex`)
   - Character Strings (`character*n`)
   - One-Dimensional Arrays (`integer a(9)`)

- Two-Dimensional Arrays (`integer a(4,4)`)
- Structured Records (`structure` & `record`)
- Pointers

For the function return values, there is an example for each of these:

- Integer (`int`)
- Real (`float`)
- Pointer to real (pointer to `float`)
- Double precision (`double`)
- Complex
- Character string

Sample Interface

In Code Example 8-1, a Fortran `main` calls a C++ function. Both `i` and `f` are references.

Code Example 8-1 Sample C++—Fortran Interface

`Samp.cc`	
	```
	extern "C" void Samp ( int &i, float& f ) {
	    i = 9;
	    f = 9.9;
	}
	```

`Sampmain.f`	
	```
	      integer i
	      real r
	      external Samp !$pragma C ( Samp )
	      call Samp ( i, r )
	      write( *, "(I2, F4.1)" ) i, r
	      end
	```

Here, both `i` and `r` are passed by reference to the default. The following command lines compile and execute `Samp.cc`, with output:

```
% CC -c Samp.cc
% f77 -c -silent Sampmain.f
% f77 Samp.o Sampmain.o -Bstatic -lC -Bdynamic
% a.out
 9 9.9
```

Compatibility Issues

Most C++ and Fortran interfaces must be correct in the following:

- Definition and call of function and subroutine
- Compatibility of data types
- Passing arguments by reference or value
- C++ function names must be uppercase and/or lowercase with a trailing underscore (_) unless the C () pragma is used in the Fortran program
- Using Fortran libraries to link

Some C++ and Fortran interfaces must also be correct on these constructs:

- Indexing and order of arrays
- File descriptors and `stdio`
- File permissions

Function versus Subroutine

The word *function* means different things in C++ and Fortran.

- As far as C++ is concerned, all subroutines are functions; the difference is that some functions return a null value.

- As far as Fortran is concerned, a function passes a return value; a subroutine does not.

Fortran Calls C++

If the C++ function returns a value, call it from Fortran as a function. If it does not return a value, call it as a subroutine.

C++ Calls Fortran

Call a Fortran function from C++ as a function. Call a Fortran subroutine from C++ as a function that returns a value of `int` (comparable to Fortran `INTEGER*4`) or `void`. This return value is useful if the Fortran routine does a nonstandard return.

Data Type Compatibility

Table 8-1 shows the default data type sizes and alignments (that is, without `-f`, `-i2`, `-misalign`, `-r4`, or `-r8`).

Note – In the following table, `REAL*16` and `COMPLEX*32` can be passed between Fortran and C++, but not between Fortran and C++ versions prior to C++ 4.0.

Table 8-1 Argument Sizes and Alignments, Passed by Reference

Fortran Type	**C++ Type**	**Size (bytes)**	**Alignment (bytes)**
`byte x`	`char x`	1	1
`character x`	`char x`	1	1
`character*`*`n`* `x`	`char x[n]`	n	1
`complex x`	`struct {float r,i;} x;`	8	4
`complex*8 x`	`struct {float r,i;} x;`	8	4
`double complex x`	`struct {double dr,di;}x;`	16	4
`complex*16 x`	`struct {double dr,di;}x;`	16	4
`double precision x`	`double x`	8	4
`real x`	`float x`	4	4
`real*4 x`	`float x`	4	4
`real*8 x`	`double x`	8	4
`integer x`	`int x`	4	4
`integer*2 x`	`short x`	2	2
`integer*4 x`	`int x`	4	4
`logical x`	`int x`	4	4
`logical*4 x`	`int x`	4	4

Table 8-1 Argument Sizes and Alignments, Passed by Reference *(Continued)*

Fortran Type	**C++ Type**	**Size (bytes)**	**Alignment (bytes)**
`logical*2 x`	`short x`	2	2
logical*1 x	char x	1	1

Note that:

- Alignments are for Fortran types.
- Arrays pass by reference, if the elements are compatible.
- Structures pass by reference, if the fields are compatible.
- When passing arguments by value:
  - You cannot pass arrays, character strings, or structures by value.
  - You can pass arguments by value from Fortran to C++, but not from C++ to Fortran, since the `%VAL()` does not work in a `SUBROUTINE` statement.

Arguments Passed by Reference or Value

In general, Fortran passes arguments by reference. In a call, if you enclose an argument with the nonstandard function `%VAL()`, Fortran passes it by value.

In C++, the function declaration tells whether an argument is passed by value or by reference.

Uppercase and Lowercase

C++ is case-sensitive; uppercase and lowercase have different meanings. The Fortran default is to ignore case by converting subprogram names to lowercase, except within character-string constants.

There are two common solutions to the uppercase and lowercase problem:

- In the C++ subprogram, make the name of the C++ function all lowercase in order to match Fortran default behavior.
- Compile the Fortran program with the `-U` option. Fortran then preserves existing uppercase and lowercase distinctions.

Use one of these solutions, but not both.

Most examples in this chapter use lowercase letters for the name in the C++ function; they do not use the Fortran `–U` compiler option.

Underscore in Names of Routines

The Fortran compiler normally appends an underscore (_) to the names of subprograms, for both a subprogram and a call to a subprogram. The underscore distinguishes it from C++ procedures or external variables with the same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All Fortran library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

There are two common solutions to the underscore problem:

- In the C++ function, change the name of the function by appending an underscore to that name.
- Use the `C()` pragma to instruct the Fortran compiler to omit those trailing underscores.

Use one of these solutions, but not both.

Most of the examples in this chapter use the Fortran `C()` compiler pragma and do not use the underscores.

The `C()` pragma directive takes the names of external functions as arguments. It specifies that these functions are written in the C or C++ language, so the Fortran compiler does not append an underscore to such names, as it ordinarily does with external names. The `C()` directive for a particular function must appear before the first reference to that function. It must also appear in each subprogram that contains such a reference. The conventional usage is:

```
    EXTERNAL ABC, XYZ!$PRAGMA C( ABC, XYZ )
```

If you use this pragma, then in the C++ function do not append an underscore to external names.

C++ Name Encoding

To implement function overloading and type-safe linkage, the C++ compiler normally appends `type` information to the names of functions. To prevent the C++ compiler from appending `type` information to the names of functions, and to allow Fortran to call functions, declare C++ functions with the `extern "C"` language construct. One common way to do this is in the declaration of a function:

```
extern "C" void abc ( int, float );
...
void abc ( int x, float y ) { /* ... body of abc ... */ }
```

For brevity, you can also combine `extern "C"` with the function definition, as in:

```
extern "C" void abc ( int x, float y )
{
        /* ... body of abc ... */
}
```

Most of the C++ examples in this chapter use this combined form. You cannot use the `extern "C"` language construct for member functions.

Array Indexing and Order

C++ arrays always start at zero, but by default, Fortran arrays start at 1. There are two common ways of approaching this.

- You can use the Fortran default, as in the above example. Then the Fortran element `B(2)` is equivalent to the C++ element `b[1]`.
- You can specify that the Fortran array `B` starts at `0`.

```
    INTEGER B(0:2)
```

This way, the Fortran element `B(1)` is equivalent to the C element `b[1]`.

Fortran arrays are stored in column-major order, C++ arrays in row-major order. For one-dimensional arrays, this is no problem. For two-dimensional arrays, this is only a minor problem, as long as the array is square. Sometimes it is enough to just switch subscripts.

For two-dimensional arrays that are *not* square, it is not enough to just switch subscripts. Try passing the whole array to the other language and do all the matrix manipulation there.

File Descriptors and `stdio`

Fortran I/O channels are in terms of unit numbers. The I/O system does not deal with unit numbers, but with file descriptors. The Fortran runtime system translates from one to the other, so most Fortran programs don't have to know about file descriptors. Many C++ programs use a set of subroutines called standard I/O (or `stdio`). Many functions of Fortran I/O use standard I/O, which in turn uses operating system I/O calls.

Table 6-2 describes some of the characteristics of these I/O systems:

Table 8-2 Characteristics of Three I/O Systems

	Fortran Units	**Standard I/O File Pointers**	**File Descriptors**
Files Open	Opened for reading and writing	Opened for reading, or opened for writing, or opened for both, or opened for appending see `OPEN`(3S)	Opened for reading, or opened for writing, or opened for both
Attributes	Formatted or unformatted	Always unformatted, but can be read or written with format-interpreting routines	Always unformatted
Access	Direct or sequential	Direct access if the physical file representation is direct access, but can always be read sequentially	Direct access if the physical file representation is direct access, but can always be read sequentially
Structure	Record	Character stream	Character stream

Table 8-2 Characteristics of Three I/O Systems *(Continued)*

	Fortran Units	**Standard I/O File Pointers**	**File Descriptors**
Form	Arbitrary, nonnegative integers	Pointers to structures in the user's address space	Integers from 0-63

File Permissions

C++ programmers traditionally open input files for reading and output files for writing, sometimes for both. In Fortran, the system cannot foresee what use you will make of the file since there's no parameter to the `OPEN` statement that gives that information.

Fortran tries to `OPEN` a file with the maximum permissions possible—first for both reading and writing, then for each separately.

This process takes place transparently and should be of concern only if you try to perform a `READ`, `WRITE`, or `ENDFILE`, when you do not have permission to do so. Magnetic tape operations are an exception to this general freedom, since you could have write permissions on a file but not a write ring on the tape.

Fortran Calls C++

This section describes the interface when Fortran calls C++.

Arguments Passed by Reference

There are two types of arguments: simple types and complex types.

Simple Types

For simple types, define each C++ argument as a reference.

Code Example 8-2 Passing Arguments by Reference—C++ Program

SimRef.cc	
	```
	extern "C" void simref (
	    char& t,
	    char& f,
	    char& c,
	    int& i,
	    float& r,
	    double& d,
	    short& si )
	{
	    t = 1;
	    f = 0;
	    c = 'z';
	    i = 9;
	    r = 9.9;
	    d = 9.9;
	    si = 9;
	}
	```

Default: Pass each Fortran argument by reference.

Code Example 8-3 Passing Arguments by Reference—Fortran Program

SimRefmain.f	
	```
	      logical*1 t, f
	      character c
	      integer*4 i
	      real r
	      double precision d
	      integer*2 si
	      external SimRef !$pragma C( SimRef )
	      call SimRef ( t, f, c, i, r, d, si )
	      write( *, "(L2,L2,A2,I2,F4.1,F4.1,I2)" )
	&    t,f,c,i,r,d,si
	      end
	```

Compile and execute, with output, as follows:

```
% CC -c SimRef.cc
% f77 -c -silent SimRefmain.f
% f77 SimRef.o SimRefmain.o -Bstatic -lC -Bdynamic
% a.out
 T F z 9 9.9 9.9 9
```

Complex Types

Here, the C++ argument is a pointer to a structure.

Code Example 8-4 Passing Arguments by Reference—Fortran Calls C++

CmplxRef.cc	`struct complex { float r, i; };`
	`struct dcomplex { double r, i; };`
	
	`extern "C" void cmplxref ( complex& w, dcomplex& z ) {`
	`    w.r = 6;`
	`    w.i = 7;`
	`    z.r = 8;`
	`    z.i = 9;`
	`}`

CmplxRefmain.f	`      complex w`
	`      double complex z`
	`      external CmplxRef !$pragma C ( CmplxRef )`
	`      call CmplxRef( w, z )`
	`      write(*,*) w`
	`      write(*,*) z`
	`      end`

Compile and execute, with output.

```
% CC -c CmplxRef.cc
% f77 -c -silent CmplxRefmain.f
% f77 CmplxRef.o CmplxRefmain.o -Bstatic -lC -Bdynamic
% a.out
  ( 6.00000, 7.00000)
  ( 8.0000000000000, 9.0000000000000)
```

A C++ reference to a `float` matches a `REAL` passed by reference.

Character Strings Passed by Reference

Passing strings between C++ and Fortran is not recommended.

For every Fortran argument of character type, Fortran associates an extra argument, giving the length of the string. The string lengths are equivalent to C++ `long int` quantities passed by value. In standard C++ use, all C++ strings are passed by reference. The order of arguments is:

- Address for each argument (datum or function)
- The length of each character argument, as a `long int`

The whole list of string lengths comes after the whole list of other arguments.

The Fortran call in:

```
    CHARACTER*7 S
    INTEGER B(3)
(arguments)
    CALL SAM( B(2), S )
```

is equivalent to the C++ call in:

```
    char s[7];
    long int b[3];
(arguments)
    sam_( &b[1], s, 7L );
```

Ignoring the Extra Arguments

You can ignore these extra arguments, since they are after the list of other arguments.

Code Example 8-5 Passing Strings by Reference—Ignoring the Extra Arguments

StrRef.cc

```
#include <string.h>

extern "C" void strref ( char (&one)[], char (&two)[] ) {
    static char letters[27] = "abcdefghijklmnopqrstuvwxyz";

    strncpy( one, letters, 10 );
    strncpy( two, letters, 26 );
}
```

Code Example 8-5 Passing Strings by Reference—Ignoring the Extra Arguments

StrRefmain.f

```
      character s10*10, s26*26
      external StrRef !$pragma C( StrRef )
      Call StrRef( s10, s26 )
      write( *, 1 ) s10, s26
 1    format( "s10='", A, "'", / "s26='", A, "'" )
      end
```

Compile and execute, with output:

```
% CC -c StrRef.cc
% f77 -c -silent StrRefmain.f
% f77 StrRef.o StrRefmain.o -Bstatic -lC -Bdynamic
% a.out
s10='abcdefghij'
s26='abcdefghijklmnopqrstuvwxyz'
```

Using the Extra Arguments

You can use the extra arguments. In the following example, the C++ function uses the lengths to match the actual arguments:

Code Example 8-6 Passing Strings by Reference—Using the Extra Arguments

StrRef2.cc

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

extern "C" void strref ( char (&one)[], char (&two)[],
int one_len, int two_len ) {
    static char letters[27] = "abcdefghijklmnopqrstuvwxyz";
    printf( "%d %d\n", L10, L26 );
    strncpy( one, letters, one_len );
    strncpy( two, letters, two_len );
}
```

Compile and execute, with output:

```
% CC -c StrRef2.cc
% f77 -c -silent StrRefmain.f
% f77 StrRef2.o StrRefmain.o -Bstatic -lC -Bdynamic
% a.out
10 26
s10='abcdefghij'
s26='abcdefghijklmnopqrstuvwxyz'
```

One-Dimensional Arrays Passed by Reference

Code Example 8-7 shows a C++ array, indexed from 0 through 8:

Code Example 8-7 Passing Arrays by Reference (C++ code)

FixVec.cc

```
extern "C" void fixvec ( int V[9], int& Sum )
{
    Sum= 0;
    for( int i= 0; i < 9; ++i ) {
        Sum += V[i];
    }
}
```

Code Example 8-8 shows a Fortran array, implicitly indexed from 1 through 9:

Code Example 8-8 Passing Arrays by Reference (Fortran code)

FixVecmain.f

```
      integer i, Sum
      integer a(9) / 1,2,3,4,5,6,7,8,9 /
      external FixVec !$pragma C( FixVec )
      call FixVec( a, Sum )
      write( *, '(9I2, " ->" I3)') (a(i),i=1,9), Sum
      end
```

Compile and execute, with output:

```
% CC -c FixVec.cc
% f77 -c -silent FixVecmain.f
% f77 FixVec.o FixVecmain.o -Bstatic -lC -Bdynamic
% a.out
 1 2 3 4 5 6 7 8 9 -> 45
```

Two-Dimensional Arrays Passed by Reference

In a two-dimensional array, the rows and columns are switched. Such arrays are either incompatible between C++ and Fortran, or awkward to keep straight. Non-square arrays are even more difficult to maintain.

Code Example 8-9 shows a 2 by 2 C++ array, indexed from 0 to 1, and 0 to 1.

Code Example 8-9 A Two-Dimensional C++ Array

FixMat.cc

```
extern "C" void fixmat ( int a[2][2] )
{
    a[0][1] = 99;
}
```

Code Example 8-10 shows a 2 by 2 Fortran array, explicitly indexed from 0 to 1, and 0 to 1.

Code Example 8-10 A Two-Dimensional Fortran Array

FixMatmain.f

```
      integer c, m(0:1,0:1) / 00, 10, 01, 11 /, r
      external FixMat !$pragma C( FixMat )
      do r= 0, 1
         do c= 0, 1
            write( *, '("m(",I1,",",I1,")=",I2.2)')  r, c, m(r,c)
         end do
      end do

      call FixMat( m )
      write( *, * )

      do r= 0, 1
         do c= 0, 1
            write( *, '("m(",I1,",",I1,")=",I2.2)')  r, c, m(r,c)
         end do
      end do

      end
```

Compile and execute. Show m before and after the C call.

```
% CC -c FixMat.cc
% f77 -c -silent FixMatmain.f
% f77 FixMat.o FixMatmain.o -Bstatic -lC -Bdynamic
% a.out
m(0,0) = 00
m(0,1) = 01
m(1,0) = 10
m(1,1) = 11

m(0,0) = 00
m(0,1) = 01
m(1,0) = 99
m(1,1) = 11
```

Compare `a[0][1]` with `m(1,0)`: C++ changes `a[0][1]`, which is Fortran `m(1,0)`.

Structured Records Passed by Reference

Code Example 8-11 and Code Example 8-12 show how to pass a structure to Fortran:

Code Example 8-11 Passing Structures to Fortran (C++ Code)

StruRef.cc

```
struct VarLenStr {
    int nbytes ;
    char a[26];
};

#include <stdlib.h>
#include <string.h>
extern "C" void struchr ( VarLenStr& v )
{
    memcpy(v.a, "oyvay", 5);
    v.nbytes= 5;
}
```

Code Example 8-12 Passing Structures to Fortran (Fortran Code)

StruRefmain.f

```
      structure /VarLenStr/
          integer nbytes
          character a*25
      end structure
      record /VarLenStr/ vls
      character s25*25

      external StruChr !$pragma C(StruChr)

      vls.nbytes= 0
      call StruChr( vls )
      s25(1:5) = vls.a(1:vls.nbytes)
      write(*, 1 ) vls.nbytes, s25
 1    format( "size =", I2, ", s25='", A, "'" )
      end
```

Compile and execute, with output:

```
% CC -c StruRef.cc
% f77 -c -silent StruRefmain.f
% f77 StruRef.o StruRefmain.o -Bstatic -lC -Bdynamic
% a.out
size = 5, s25='oyvay'
```

Pointers Passed by Reference

C++ gets a reference to a pointer, as follows:

Code Example 8-13 Passing Pointers by Reference (C++ Code)

PassPtr.cc

```
extern "C" void passptr ( int* & i, double* & d )
{
    *i = 9;
    *d = 9.9;
}
```

Code Example 8-14 shows how Fortran passes the pointer by reference:

Code Example 8-14 Passing Pointers by Reference (Fortran code)

PassPtrmain.f	

	`program PassPtrmain`
	`integer i`
	`double precision d`
	`pointer (iPtr, i), (dPtr, d)`
	`external PassPtr !$pragma C ( PassPtr )`
	`iPtr = malloc( 4 )`
	`dPtr = malloc( 8 )`
	`i = 0`
	`d = 0.0`
	`call PassPtr( iPtr, dPtr )`
	`write( *, "(i2, f4.1)" ) i, d`
	`end`

Compile and execute, with output:

```
% CC -c PassPtr.cc
% f77 -c -silent PassPtrmain.f
% f77 PassPtr.o PassPtrmain.o -Bstatic -lC -Bdynamic
% a.out
 9 9.9
```

Arguments Passed by Value

In the call, enclose an argument in the nonstandard function `%VAL()`.

Simple Types Passed by Value

Code Example 8-15 and Code Example 8-16 show how to pass simple types by value.

Code Example 8-15 Passing Simple Types by Value (C++ Code)

SimVal.cc

```
extern "C" void simval (
    char   t,
    char   f,
    char   c,
    int    i,
    double d,
    short  s,
    int&   reply )
{
    reply= 0;
    // If nth arg ok, set nth octal digit to one
    if( t        ) reply +=      01;
    if( ! f      ) reply +=     010;
    if( c == 'z' ) reply +=    0100;
    if( i == 9   ) reply +=   01000;
    if( d == 9.9 ) reply +=  010000;
    if( s == 9   ) reply += 0100000;
}
```

Code Example 8-16 Passing Simple Types by Value (Fortran Code)

SimValmain.f

```
      logical*1 t, f
      character c
      integer*4 i
      double precision d
      integer*2 s
      integer*4 args

      data t/.true./, f/.false./, c/'z'/
&    i/9/, d/9.9/, s/9/

      external SimVal !$pragma C( SimVal )
      call SimVal ( %VAL(t), %VAL(f), %VAL(c),
&        %VAL(i), %VAL(d), %VAL(s), args )
      write( *, 1 ) args
 1    format( 'args=', o6, ' (If nth digit=1, arg n OK)' )
      end
```

Pass each Fortran argument by value, except for `args`. The same rule applies to `CHARACTER*1`, `COMPLEX`, `DOUBLE COMPLEX`, `INTEGER`, `LOGICAL`, `DOUBLE PRECISION`, structures, and pointers.

Compile and execute, with output:

```
% CC -c SimVal.cc
% f77 -c -silent SimValmain.f
% f77 SimVal.o SimValmain.o -Bstatic -lC -Bdynamic
% a.out
args=111111(If nth digit=1, arg n OK)
```

Real Variables Passed by Value

Real variables are passed by value the same way other simple types are. Code Example 8-17 shows how to pass a real variable:

Code Example 8-17 Passing a Real Variable

FloatVal.cc

```
#include <math.h>

extern "C" void floatval ( float f, double& d ) {
    float x=f;
    d = double(x) + 1.0 ;
}
```

FloatValmain.f

```
      double precision d
      real r / 8.0 /
      external FloatVal !$pragma C( FloatVal )
      call FloatVal( %VAL(r), d )
      write( *, * ) r, d
      end
```

Compile and execute, with output:

```
% CC -c FloatVal.cc
% f77 -c -silent FloatValmain.f
% f77 FloatVal.o FloatValmain.o -Bstatic -lC -Bdynamic
% a.out
    8.00000 9.0000000000000
```

Complex Types Passed by Value

You can pass the `complex` structure by value, as Code Example 8-18 shows:

Code Example 8-18 Passing Complex Types

`CmplxVal.cc`	
	```
	struct complex { float r, i; };

	extern "C" void cmplxval ( complex  w, complex& z ) {
	    z.r = w.r * 2.0 ;
	    z.i = w.i * 2.0 ;
	    w.r = 0.0 ;
	    w.i = 0.0 ;
	}
	```

`CmplxValmain.f`	
	```
	      complex w / (4.0, 4.5 ) /
	      complex z
	      external CmplxVal !$pragma C( CmplxVal )
	      call CmplxVal( %VAL(w), z )
	      write ( *, * ) w
	      write ( *, * ) z
	      end
	```

Compile and execute, with output

```
% CC -c CmplxVal.cc
% f77 -c -silent CmplxValmain.f
% f77 CmplVal.o CmplxValmain.o -Bstatic -lC -Bdynamic
% a.out
  ( 4.00000, 4.50000)
  ( 8.00000, 9.00000)
```

Arrays, Strings, Structures Passed by Value

There is no reliable way to pass arrays, character strings, or structures by value on all architectures. The workaround is to pass them by reference.

Pointers Passed by Value

C++ gets a pointer.

Code Example 8-19 Passing Pointers by Value (C++ Code)

PassPtrVal.cc

```
extern "C" void passptrval ( int* i, double* d )
{
    *i = 9;
    *d = 9.9;
}
```

Fortran passes a pointer by value:

Code Example 8-20 Passing Pointers by Value (Fortran Code)

PassPtrValmain.f

```
      program PassPtrValmain
      integer i
      double precision d
      pointer (iPtr, i), (dPtr, d)
      external PassPtrVal !$pragma C ( PassPtrVal )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtrVal( %VAL(iPtr), %VAL(dPtr) ) ! Nonstandard?
      write( *, "(i2, f4.1)" ) i, d
      end
```

Compile and execute, with output:

```
% CC -c PassPtrVal.cc
% f77 -c -silent PassPtrValmain.f
% f77 PassPtrVal.o PassPtrValmain.o -Bstatic -lC -Bdynamic
% a.out
 9 9.9
```

Function Return Values

For function return values, a Fortran function of type BYTE, INTEGER, REAL, LOGICAL, or DOUBLE PRECISION is equivalent to a C++ function that returns the corresponding type. There are two extra arguments for the return values of character functions, and one extra argument for the return values of complex functions.

int

Code Example 8-21 shows how to return an `int` to a Fortran program.

Code Example 8-21 Returning an `int` to Fortran

`RetInt.cc`	
	```
	extern "C" int retint ( int& r )
	{
	    int s;
	    s = r;
	    ++s;
	    return s;
	}
	```

`RetIntmain.f`	
	```
	      integer r, s, RetInt
	      external RetInt !$pragma C( RetInt )
	      r = 2
	      s = RetInt( r )
	      write( *, "(2I4)") r, s
	      end
	```

Compile, link, and execute, with output.

```
% CC -c RetInt.cc
% f77 -c -silent RetInt.o RetIntmain.f
% f77 RetInt.o RetIntmain.o -Bstatic -lC -Bdynamic
% a.out
 2 3
%
```

Do a function of type `BYTE`, `LOGICAL`, `REAL`, or `DOUBLE PRECISION` in the same way. Use matching types according to Table 8-1 on page 124.

float

Code Example 8-22 shows how to return a `float` to a Fortran program:

Code Example 8-22 Return a `float` to Fortran

RetFloat.cc

```
extern "C" float retfloat ( float& pf )
{
    float f;
    f = pf;
    ++f;
    return f;
}
```

RetFloatmain.f

```
      real RetFloat, r, s
      external RetFloat !$pragma C( RetFloat )
      r = 8.0
      s = RetFloat( r )
      print *, r, s
      end
```

```
% CC -c RetFloat.cc
% f77 -c -silent RetFloatmain.f
% f77 RetFloat.o RetFloatmain.o -Bstatic -lC -Bdynamic
% a.out
    8.00000 9.00000
```

A Pointer to a `float`

Code Example 8-23 shows how to return a function value that is a pointer to a `float`.

Code Example 8-23 Return a pointer to a `float` to Fortran

RetPtrF.cc

```
static float f;

extern "C" float* retptrf ( float& a )
{
    f = a;
    ++f;
    return &f;
}
```

Code Example 8-23 Return a pointer to a `float` to Fortran

`RetPtrFmain.f`	
	```
	      integer RetPtrF
	      external RetPtrF !$pragma C( RetPtrF )
	      pointer (p, s)
	      real r, s
	      r = 8.0
	      p = RetPtrF( r )
	      print *, s
	      end
	```

Compile and execute, with output:

```
% CC -c RetPtrF.cc
% f77 -c -silent RetPtrFmain.f
% f77 RetPtrF.o RetPtrFmain.o -Bstatic -lC -Bdynamic
% a.out
9.00000
```

The function return value is an address; you can assign it to the pointer value, or do some pointer arithmetic. You cannot use it in an expression with `real`s, such as `RetPtrF(R)+100.0`.

Double Precision

Code Example 8-24 is an example of C++ returning a type `double` function value to a Fortran `DOUBLE PRECISION` variable:

Code Example 8-24 Return a `double` to Fortran

`RetDbl.cc`	
	```
	extern "C" double retdbl ( double& r )
	{
	    double s;
	    s = r;
	    ++s;
	    return s;
	}
	```

Code Example 8-24 Return a double to Fortran

RetDblmain.f

```
      double precision r, s, RetDbl
      external RetDbl !$pragma C( RetDbl )
      r = 8.0
      s = RetDbl( r )
      write( *, "(2F6.1)" ) r, s
      end
```

Compile and execute, with output.

```
% CC -c RetDbl.cc
% f77 -c -silent RetDblmain.f
% f77 RetDbl.o RetDblmain.o -Bstatic -lC -Bdynamic
% a.out
   8.0 9.0
```

COMPLEX

A COMPLEX or DOUBLE COMPLEX function is equivalent to a C++ routine having an additional initial argument that points to the return value storage location. A general pattern for such a Fortran function is:

```
     COMPLEX FUNCTION F ( arguments )
```

The pattern for a corresponding C++ function is:

```
struct complex { float r, i; };
f_ ( complex temp, arguments );
```

Code Example 8-25 shows how to return a type COMPLEX function value to Fortran.

Code Example 8-25 Returning a COMPLEX value to Fortran

RetCmplx.cc

```
struct complex { float r, i; };

extern "C" void retcmplx ( complex& RetVal, complex& w ) {
    RetVal.r = w.r + 1.0 ;
    RetVal.i = w.i + 1.0 ;
    return;
}
```

Code Example 8-25 Returning a COMPLEX value to Fortran *(Continued)*

`RetCmplxmain.f`	`complex u, v, RetCmplx`
	`external RetCmplx !$pragma C( RetCmplx )`
	`u = ( 7.0, 8.0 )`
	`v = RetCmplx( u )`
	`write( *, * ) u`
	`write( *, * ) v`
	`end`

Compile and execute, with output:

```
% CC -c -silent RetCmplx.cc
% f77 -c -silent RetCmplxmain.f
% f77 RetCmplx.o RetCmplxmain.o -Bstatic -lC -Bdynamic
% a.out
  ( 7.00000, 8.00000)
  ( 8.00000, 9.00000)
```

Character Strings

Passing strings between C++ and Fortran is not recommended. A character-string-valued Fortran function is equivalent to a C++ function with the two extra initial arguments—data address and length. A Fortran function of this form:

```
    CHARACTER*15 FUNCTION G (arguments)
```

and a C++ function of this form:

```
g_ (char * result, long int length, other arguments)
char result[ ];
long int length;
```

are equivalent and can be invoked in C++ with this call:

```
char chars[15];
(other arguments)
    g_ (chars, 15L, other arguments);
```

Code Example 8-26 shows how to return a character string to a Fortran program.

Code Example 8-26 Returning a String to Fortran (C++ Code)

RetStr.cc

```
#include <stdio.h>

extern "C" void retstr_ ( char *retval_ptr, int retval_len,
    char& ch_ref,
    int& n_ref,
    int  ch_len )
{
    int count = n_ref;
    char *cp = retval_ptr;
    for( int i= 0; i < count; ++i ) {
    *cp++ = ch_ref;
    }
}
```

- The returned string is passed by the extra arguments `retval_ptr` and `retval_len`, a pointer to the start of the string and the string's length.
- The character-string argument is passed with `ch_ref` and `ch_len`.
- The `ch_len` is at the end of the argument list.
- The repeat factor is passed as `n_ref`.

In Fortran, use the preceding C++ function as shown here:

Code Example 8-27 Returning a String to Fortran (Fortran Code)

RetStrmain.f

```
      character String*100, RetStr*50
      String = RetStr( '*', 10 )
      print *, "'", String(1:10), "'"
      end
```

Compile and execute with output:

```
% CC -c RetStr.cc
% f77 -c -silent RetStrmain.f
% f77 RetStr.o RetStrmain.o -Bstatic -lC -Bdynamic
% a.out
'**********'
```

Labeled Common

C++ and Fortran can share values in labeled common. The method is the same no matter which language calls the other, as shown by Code Example 8-28 and Code Example 8-29:

Code Example 8-28 Using Labeled Common (Fortran Code)

UseCom.cc

```
#include <stdio.h>
#include <stdlib.h>
struct ilk_type {
    float p;
    float q;
    float r;
};

extern ilk_type ilk_;
extern "C" void usecom_ ( int& count ){
    ilk_.p = 7.0;
    ilk_.q = 8.0;
    ilk_.r = 9.0;
}
```

Code Example 8-29 Using Labeled Common (C++ Code)

UseCommain.f

```
integer n
real u, v, w
common /ilk/ u, v, w
    n = 3
    u = 1.0
    v = 2.0
    w = 3.0
call usecom (n)
print *, u, v, w
end
```

Compile and execute, with output:

```
% f77 -c -silent UseCom.f
% CC -c UseCommain.cc
% f77 UseCom.o UseCommain.o -Bstatic -lC -Bdynamic
% a.out
 ilk_p = 7.0, ilk_q = 8.0, ilk_r = 9.0
```

Any of the options that change size or alignment, or any equivalences that change alignment, may invalidate such sharing.

I/O Sharing

It’s not a good idea to mix Fortran I/O with C++ I/O. It’s safer to pick one and not alternate.

The Fortran I/O library is implemented largely on top of the C standard I/O library. Every open unit in a Fortran program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them. However, the C++ stream I/O system uses a different mechanism.

If a Fortran main program calls C++, then before the Fortran program starts, the Fortran I/O library is initialized to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. The C++ function must take the Fortran I/O environment into consideration to perform I/O on open file descriptors.

stdout

Code Example 8-30 shows a C++ function that writes to `stderr` and to `stdout`, and the Fortran code that calls the C++ function:

Code Example 8-30 Mixing with `stdout`

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

	extern "C" void mixio ( int& n ) {
	    if( n <= 0 ) {
	    fprintf( stderr, "Error:  negative line number (%d)\n", n );
	    n= 1;
	    }
	    printf( "In   C++:    line # = %2d\n", n );
	}
	```

`MixIOmain.f`	
	```
	      integer n/ -9 /
	      external MixIO !$pragma C( MixIO )
	      do i= 1, 6
	         n = n +1
	         if ( abs(mod(n,2)) .eq. 1 ) then
	            call MixIO( n )
	         else
	            write( *, '("In Fortran:  line # = ", i2)' ) n
	         end if
	      end do
	      end
	```

Compile and execute, with output:

```
% CC -c MixIO.cc
% f77 -c -silent MixIOmain.f
% f77 MixIO.o MixIOmain.o -Bstatic -lC -Bdynamic
% a.out
In Fortran: line # =-8
error: negative line #
In C: line # = 1
In Fortran: line # = 2
In C: line # = 3
In Fortran: line # = 4
In C: line # = 5
```

`stdin`

Code Example 8-31 shows a C++ function that reads from `stdin`, and the Fortran code that calls the C++ function:

Code Example 8-31 Mixing with `stdin`

MixStdin.cc

```
#include <stdio.h>

extern "C" int c_read_ ( FILE* &fp, char *buf, int& nbytes, int
buf_len )
{
    return fread( buf, 1, nbytes, fp );
}
```

MixStdinmain.f

```
      character*1 inbyte
      integer*4 c_read, getfilep
      external getfilep
      write( *, '(a, $)') 'What is the digit? '
      flush (6)
      irtn = c_read( getfilep( 5 ), inbyte, 1 )
      write( *, 9 ) inbyte
 9    format( 'The digit read by C++ is ', a )
      end
```

Fortran does the prompt; C++ does the read, as follows:

```
% CC -c MixStdin.cc
% f77 -c -silent MixStdinmain.f
% f77 MixSdin.o MixStdinmain.o -Bstatic -lC -Bdynamic
% a.out
What is the digit? 3
The digit read by C is 3
demo%
```

Alternate Returns

C++ does not have an alternate return. The workaround is to pass an argument and branch on that.

C++ Calls Fortran

This section describes the interface when C++ calls Fortran.

Arguments Passed by Reference

Simple Variables Passed by Reference

Here, Fortran expects all these arguments to be passed by reference, which is the default.

Code Example 8-32 Passing Variables by Reference (Fortran Code)

SimRef.f

```
       subroutine SimRef ( t, f, c, i, d, si, sr )
       logical*1 t, f
       character c
       integer i
       double precision d
       integer*2 si
       real sr
       t = .true.
       f = .false.
       c = 'z'
       i = 9
       d = 9.9
       si = 9
       sr = 9.9
       return
       end
```

Here, C++ passes the address of each.

Code Example 8-33 Passing Variables by Reference (C++ Code)

SimRefmain.cc

```
extern "C" void simref_( char&, char&, char&, int&, double&,
short&, float& );
#include <stdio.h>
#include <stdlib.h>

main ( ) {
    char t, f, c;
    int i;
    double d;
    short si;
    float sr;

    simref_( t, f, c, i, d, si, sr );
    printf( "%08o %08o %c %d %3.1f %d %3.1f\n",
       t, f, c, i, d, si, sr );
    return 0;
}
```

Compile and execute, with output:

```
% f77 -c -silent SimRef.f
% CC -c SimRefmain.cc
% f77 SimRef.o SimRefmain.o -Bstatic -lC -Bdynamic
% a.out
00000001 00000000 z 9 9.9 9 9.9
demo%
```

Complex Variables Passed by Reference

The complex types require a simple structure as shown in Code Example 8-34. In this example `w` and `z` are passed by reference, which is the default:

Code Example 8-34 Passing Complex Variables

`CmplxRef.f`	
	```
	      subroutine CmplxRef ( w, z )
	      complex w
	      double complex z
	      w = ( 6, 7 )
	      z = ( 8, 9 )
	      return
	      end
	```

`CmplxRefmain.cc`	
	```
	#include <stdlib.h>
	#include <stdio.h>

	struct complex { float r, i; };
	struct dcomplex { double r, i; };

	extern "C" void cmplxref_ ( complex& w, dcomplex& z );

	main ( ) {
	    complex d1;
	    dcomplex d2;

	    cmplxref_( d1, d2 );
	   printf( "%3.1f %3.1f\n%3.1f %3.1f\n", d1.r, d1.i, d2.r, d2.i );
	    return 0;
	}
	```

The following example shows `CmplxRef.f` compiled and executed with output:

```
% f77 -c -silent CmplxRef.f
% CC -c CmplxRefmain.cc
% f77 CmplxRef.o CmplxRefmain.o -Bstatic -lC -Bdynamic
% a.out
6.0 7.0
8.0 9.0
```

Character Strings Passed by Reference

Character strings match in a straightforward manner. If you make the string in Fortran, you must provide the explicit null terminator. Fortran does not automatically provide the terminator, and C++ expects it.

Note – Avoid passing strings between C++ and Fortran.

Code Example 8-35 Passing Strings by Reference

`StrRef.f`	
	```
	      subroutine StrRef ( a, s )
	      character a*10, s*80
	      a = 'abcdefghi' // char(0)
	      s = 'abcdefghijklmnopqrstuvwxyz' // char(0)
	      return
	      end
	```

`StrRefmain.cc`	
	```
	#include <stdlib.h>
	#include <stdio.h>

	extern "C" void strref_ ( char*, char* );

	main ( ) {
	    char s10[10], s80[80];

	    strref_( s10, s80 );
	    printf( " s10='%s'\n s80='%s'\n", s10, s80 );
	    return 0;
	}
	```

Compile and execute, with output:

```
% f77 -c -silent StrRef.f
% CC -c StrRefmain.cc
% f77 StrRef.o StrRefmain.o -Bstatic -lC -Bdynamic
% a.out
s10='abcdefghi'
s80='abcdefghijklmnopqrstuvwxyz'
```

Arguments Passed by Value

Fortran can call C++ and pass an argument by value. However Fortran cannot handle an argument passed by value. The workaround is to pass all arguments by reference.

Function Return Values

For function return values, a Fortran function of type BYTE, INTEGER, LOGICAL, or DOUBLE PRECISION is equivalent to a C++ function that returns the corresponding type. There are two extra arguments for the return values of character functions and one extra argument for the return values of complex functions.

int

Code Example 8-36 shows how Fortran returns an INTEGER function value to C++:

Code Example 8-36 Returning an Integer to C++

RetInt.f	
	```
	      integer function RetInt ( k )
	      integer k
	      RetInt = k + 1
	      return
	      end
	```

RetIntmain.cc	
	```
	#include <stdio.h>
	#include <stdlib.h>

	extern "C" int retint_ ( int& );

	main ( ) {
	      int k = 8;
	      int m = retint_( k );
	      printf( "%d %d\n", k, m );
	      return 0;
	}
	```

Compile and execute, with output:

```
% f77 -c -silent RetInt.f
% CC -c RetIntmain.cc
% f77 RetInt.o RetIntmain.o -Bstatic -lC -Bdynamic
% a.out
8 9
```

`float`

Code Example 8-37 shows how to return a `float` to a C++ program.

Code Example 8-37 Returning a `float` to C++

`RetFloat.f`	
	```
	      real function RetReal ( x )
	      real x
	      RetReal = x + 1.0
	      return
	      end
	```

`RetFloatmain.cc`	
	```
	#include <stdio.h>
	extern "C" float retreal_ (float*) ;
	main ( )
	{
	        float r, s ;
	        r = 8.0 ;
	        s = retreal_ ( &r ) ;
	        printf( " %8.6f %8.6f \n", r, s ) ;
	        return 0;
	}
	```

Compile and execute, with output:

```
% f77 -c -silent RetFloat.f
% CC -c -w RetFloatmain.cc
% f77 RetFloat.o RetFloatmain.o -Bstatic -lC -Bdynamic
% a.out
 8.000000 9.000000
```

`double`

Code Example 8-38 shows how Fortran returns a `DOUBLE PRECISION` function value to C++.

Code Example 8-38 Returning a `double` to C++

`RetDbl.f`	`      double precision function RetDbl ( x )`
	`      double precision x`
	`      RetDbl = x + 1.0`
	`      return`
	`      end`

`RetDblmain.cc`	`#include <stdio.h>`
	`#include <stdlib.h>`
	
	`extern "C" double retdbl_ ( double& );`
	
	`main ( ) {`
	`      double x = 8.0;`
	`      double y = retdbl_( x );`
	`      printf( "%8.6f %8.6f\n", x, y );`
	`      return 0;`
	`}`

Compile and execute, with output:

```
% f77 -c -silent RetDbl.f
% CC -c RetDblmain.cc
% f77 RetDbl.o RetDblmain.o -Bstatic -lC -Bdynamic
% a.out
8.000000 9.000000
```

`COMPLEX` *or* `DOUBLE COMPLEX`

A `COMPLEX` or `DOUBLE COMPLEX` function is equivalent to a C++ routine having an additional initial argument that points to the return value storage location. A general pattern for such a Fortran function is:

```
    COMPLEX FUNCTION F ( arguments )
```

The pattern for a corresponding C++ function is:

```
struct complex { float r, i; };
void f_ ( complex &, other arguments )
```

Code Example 8-39 shows how to return a COMPLEX:

Code Example 8-39 Returning a COMPLEX

`RetCmplx.f`	
	```
	      complex function RetCmplx ( x )
	      complex x
	      RetCmplx = x * 2.0
	      return
	      end
	```

`RetCmplxmain.cc`	
	```
	#include <stdlib.h>
	#include <stdio.h>
	struct complex { float r, i; };

	extern "C" void retcmplx_( complex&, complex& );

	main ( ) {
	    complex c1, c2 ;

	    c1.r = 4.0;
	    c1.i = 4.5;

	    retcmplx_( c2, c1 );

	   printf( " %3.1f %3.1f\n %3.1f %3.1f\n", c1.r, c1.i, c2.r, c2.i
	);
	    return 0;
	}
	```

Compile, link, and execute, with output:

```
% f77 -c -silent RetCmplx.f
% CC -c -w RetCmplxmain.cc
% f77 RetCmplx.o RetCmplxmain.o -Bstatic -lC -Bdynamic
% a.out
 4.0 4.5
 8.0 9.0
```

When you use `f77` to pass files to the linker, the linker uses the `f77` libraries.

Character Strings

Note – Avoid passing strings between C++ and Fortran.

A Fortran string function has two extra initial arguments: data address and length. If you have a Fortran function of the following form:

```
    CHARACTER*15 FUNCTION G ( arguments )
```

and a C++ function of this form:

```
g_ ( char * result, long int length, other arguments )
```

they are equivalent, and can be invoked in C++ with:

```
char chars[15];
g_ ( chars, 15L, arguments );
```

The lengths are passed by value. You must provide the null terminator. Code Example 8-40 shows how to pass a string to C++.

Code Example 8-40 Returning a String to C++

`RetChr.f`	
	```
	      function RetChr( c, n )
	      character RetChr*(*), c
	      RetChr = ''
	      do i = 1, n
	         RetChr(i:i) = c
	      end do

	      RetChr(n+1:n+1) = char(0) ! Put in the null terminator.
	      return
	      end
	```

Code Example 8-40 Returning a String to C++ *(Continued)*

RetChrmain.cc

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

extern "C" void retchr_( char*, int, char*, int&, int );

main ( ) {
    char string[100], repeat_val[50];

    int repeat_len = sizeof( repeat_val );
    int count = 10;

    retchr_( repeat_val, repeat_len, "*", count, sizeof("*")-1 );

    strncpy( string, repeat_val, repeat_len );
    printf( " '%s'\n", repeat_val );
    return 0;
}
```

Compile, link, and execute, with output.

```
% f77 -c -silent RetChr.f
% CC -c RetChrmain.cc
% f77 RetChr.o RetChrmain.o -Bstatic -lC -Bdynamic
% a.out
 '**********'
```

The caller must set up more arguments than are apparent as formal parameters to the Fortran function. Arguments that are lengths of character strings are passed by value. Those that are not are passed by reference.

Labeled Common

C++ and Fortran can share values in labeled common. Any of the options that change size or alignment, or any equivalences that change alignment, may invalidate such sharing.

The method is the same, no matter which language calls the other.

Code Example 8-41 Using Labeled Common (Fortran Code)

UseCom.f

```
      subroutine UseCom ( n )
      integer n
      real u, v, w
      common /ilk/ u, v, w
      n = 3
      u = 7.0
      v = 8.0
      w = 9.0
      return
      end
```

Code Example 8-42 Using Labeled Common (C++ Code)

UseCommain.cc

```
#include <stdio.h>

struct ilk_type {
    float p;
    float q;
    float r;
};

extern ilk_type ilk_ ;
extern "C" void usecom_ ( int& );

main ( ) {
    char *string = "abc0" ;
    int count = 3;
    ilk_.p = 1.0;
    ilk_.q = 2.0;
    ilk_.r = 3.0;
    usecom_( count );
    printf( " ilk_.p=%4.1f,  ilk_.q=%4.1f,  ilk_.r=%4.1f\n",
       ilk_.p, ilk_.q, ilk_.r );
    return 0;
}
```

Compile and execute, with output:

```
% f77 -c -silent UseCom.f
% CC -c UseCommain.cc
% f77 UseCom.o UseCommain.o -Bstatic -lC -Bdynamic
% a.out
 ilk_.p = 7.0, ilk_.q = 8.0, ilk_.r = 9.0
```

I/O Sharing

Avoid mixing Fortran I/O with C++ I/O. It is usually safer to pick one and not alternate.

The Fortran I/O library uses the C++ standard I/O library. Every open unit in a Fortran program has an associated standard I/O file structure. For the `stdin`, `stdout`, and `stderr` streams, the file structure need not be explicitly referenced, so it is possible to share them.

If a C++ main program calls a Fortran subprogram, there is no automatic initialization of the Fortran I/O library (connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively). If a Fortran function attempts to reference the `stderr` stream (unit 0), then any output is written to a file named `fort.0` instead of to the `stderr` stream.

To make the C++ program initialize I/O and establish the preconnection of units 0, 5, and 6, insert the following line at the start of the C++ `main`.

```
    f_init();
```

At the end of the C++ `main`, you can insert:

```
    f_exit();
```

although it may not be necessary.

Code Example 8-43 and Code Example 8-44 show how to share I/O using a C++ main program and a Fortran subroutine.

Code Example 8-43 Sharing I/O (Fortran Code)

MixIO.f

```
      subroutine MixIO ( n )
      integer n
      if ( n .le. 0 ) then
         write(0,*) "error: negative line #"
         n = 1
      end if

      write( *, '("In Fortran:  line # = ", i2 )' ) n
      end
```

Code Example 8-44 Sharing I/O (C++ Code)

MixIOmain.cc

```
#include <stdio.h>

extern "C" {
    void mixio_( int& );
    void f_init();
    void f_exit();
};

main ( ) {
    f_init();
    int m= -9;

    for( int i= 0; i < 5; ++i ) {
    ++m;
    if( m == 2  ||  m == 4 ) {
        printf( "In  C++   :  line # = %d\n", m );
    } else {
        mixio_( m );
    }
    }
    f_exit();
    return 0;
}
```

Code Example 8-45 Compile and execute, with output:

```
% f77 -c -silent MixIO.f
% CC -c -w MixIOmain.cc
% f77 MixIO.o MixIOmain.o -Bstatic -lC -Bdynamic
% a.out
error: negative line #
In Fortran: line # = 1
In C: line # = 2
In Fortran: line # = 3
In C: line # = 4
In Fortran: line # = 5
```

With a C++ `main()` program, the following Fortran library routines may not work correctly: `signal()`, `getarg()`, `iargc()`

Alternate Returns

Your C++ program may need to use a Fortran subroutine that has nonstandard returns. To C++, such subroutines return an `int` (`INTEGER*4`). The return value specifies which alternate return to use. If the subroutine has no entry points with alternate return arguments, the returned value is undefined.

Code Example 8-46 returns one regular argument and two alternate returns.

Code Example 8-46 Alternate Returns

AltRet.f

```
       subroutine AltRet ( i, *, * )
       integer i, k
       i = 9
       write( *, * ) 'k:'
       read( *, * ) k
       if( k .eq. 10 ) return 1
       if( k .eq. 20 ) return 2
       return
       end
```

Code Example 8-46 Alternate Returns *(Continued)*

AltRetmain.cc	`#include <stdio.h>`
	`#include <stdlib.h>`
	
	`extern "C" int altret_ ( int& );`
	
	`main ( ) {`
	`    int k = 0;`
	`    int m = altret_( k );`
	`    printf( "%d %d\n", k, m );`
	`    return 0;`
	`}`

C++ invokes the subroutine as a function.

Compile, link, and execute:

```
% f77 -c -silent AltRet.f
% CC -c AltRetmain.cc
% f77 AltRet.o AltRetmain.o -Bstatic -lC -Bdynamic
% a.out
k:
20
9 2
```

In this example, the C++ `main()` receives a 2 as the return value of the subroutine, because you typed in a `20`.

Migration Guide A

This section is intended to help users of earlier versions of C++ (3.0, 3.0.1, 4.0, 4.0.1 and 4.1) port their code to C++ 4.2. If you have code that compiles and runs under C++ 3.0, and would like to have it compile and run under C++ 4.2, this is, in most cases, as easy as recompiling and relinking the code.

Note – C++ 4.2 object code is not compatible with C++ 3.0 object code, so you must recompile your code, including any libraries, which might contain C++ code. C++ 4.2 object code is compatible with C++ 4.1 and 4.0 object code.

This document resides in the `README` directory; it is included in both ASCII and PostScript™ format. The PostScript file has a `.ps` suffix.

The Language

C++ is an actively changing language. There is an ANSI committee hard at work trying to generate a standard for the language, but that is still years away. In the meantime, there are the *Annotated Reference Manual* (ARM) and `C++ 3.0`, although these leave much to be desired.

When writing a compiler, you need hard facts about every aspect of the language, and when you can't find a precise definition, you have to make the best guess you can. The developers chose to follow the ARM whenever it was clear enough, and to use the ongoing work of the ANSI committee, as described in the Draft Working Paper (DWP) to help clarify murky areas. The result is their best guess at what the language will become, but you can expect changes with the next release. Sun is actively involved in the ANSI committee and the entire C++ standardization effort, so the developers are tracking its progress closely.

The compiler supports the entire language described in the ARM, including templates and exceptions. It also includes support for wide characters, including a separate type for `wchar_t`. SPARCompiler C++ 4.2 is fully integrated with Solaris 2.x internationalization features. There is an extended long integer (`long long int`) and a true `long double` type.

Sun is aware that there is a lot of code that compiles with some version of `C++ 3.0`, but is not legal under the current (or possibly any) language definition. The compiler detects and flags these errors, finding bugs, and generally improving the code quality. However, you have to compile and use your existing code, so the developers have worked very hard to allow such programs to compile, and to flag these incompatibilities with warning messages. Though changes in the language make it impossible to be completely compatible, this compiler is much more backward-compatible than earlier versions of `C++ 3.0`.

Recompiling Your Code

Run your code through the compiler. In the majority of cases, this results in anachronism warnings, indicating where the language has changed or compiler bugs have been eliminated. Examine the warnings carefully, since some may indicate undetected errors in your program. Other warnings reflect changes in the language and can be ignored until you change the program. Remove the warnings using guidelines in the following sections.

Incompatibilities: C++ 3.0 to 4.2

In some rare cases, the C++ 3.0 implementation differs from the specification in the *C++ Annotated Reference Manual* (ARM), or generates incorrect code. In these cases, discussed in this section, C++ 4.2 is incompatible with C++ 3.0, and you must modify your code.

Error: K&R-style function definitions are no longer allowed

C++ 3.0 issues warnings; C++ 4.2 issues errors.

Error: You cannot set _new_handler via an assignment

To set `_new_handler`, call `set_new_handler()` if you use your own new handler. By assigning to `_new_handler`, `ld` issues an error that `_new_handler` is undefined. Use `set_new_handler()` to set your own new handler.

Error: Multiple declarations for A

C++ 3.0 allows two arguments with the same name in function prototypes. For example

```
extern int foo (int a, int a);
```

C++ 4.2 does not allow this.

Error: Global operator new() is always used when there is no class version

Resolution of `operator new()` for nested classes in C++ 4.2 is different from C++ 3.0. C++ 3.0 erroneously uses the `operator new()` from the outer class in preference to the global `operator new()`, as specified in the ARM.

C++ 4.2 does not duplicate this bug, because to do so would change the semantics of correct programs. For example, the following test case gives different results when compiled with C++ 3.0 and C++4.2:

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

class Foo {
public:
            void *operator new(size_t sz);
    class Bar {
    public:
        int j;
        Bar(int i) {j = 1;};
    };
};

void *Foo::operator new(size_t sz) {
    printf("Hi!\n");
    return ::operator new(sz);
}

int main () {
    Foo::Bar *b = new Foo::Bar(17);
    return 0;
}
```

C++ 3.0 execution produces “Hi!” while C++ 4.2 produces no output.

Solution: In the unlikely case that your program depends on this behavior, define `Foo::Bar::operator new()` and call `Foo::operator new()` from the new function.

Error: typedef names are not structure keys

C++ 3.0 and C++ 4.2 differ in the treatment of `typedef` names for structure types. C++ 3.0 treats the `typedef` name as a `struct` name. For example, you could write:

```
typedef struct { int x; } A;
struct A avar;
```

The DWP and ARM state that the second declaration declares a new structure name A. This is an error since there is already an existing `typedef` name A. The result is:

```
Error: Multiple declaration for A.
Error: The type A is incomplete.
```

These errors show up differently when there are local scopes involved, and interpreting the code the same way as C++ 3.0 can actually change the meaning of a legal program. The following code shows some of the confusion that may arise

```
typedef struct { int x; } A;
void bug () {
    A ** ptpt;
    typedef struct A* Xss; // Declares a local struct A
    ptpt = (Xss*) new Xss[2]; // error here
    *ptpt = (A*) new(A); // errors
}
```

The error messages look like this:

```
line 5: Error: Cannot assign A(local)** to A**.
line 6: Error: The type "A(local)" is incomplete.
line 6: Error: Cannot assign A(local)* to A*.
```

Solution: Use structure tags instead of `typedef` names. To leave the `typedef` name for C compatibility, add the structure name, and use it in both places

```
typedef struct A { int x; } A;
```

Error: Redefining AAAA after use in BBBB

C++ contains rules that prohibit any redefinition of an outer scope name that has been used in the class. For example

```
typedef int TI;
class C {
    TI iv;
    float TI;
};
```

or, using the same typedef

```
class D {
    TI TI;
};
```

Both cases produce an error message:

```
Error: Redefining TI after use in C
```

Both cases are legal in C. C++ 3.0 does not detect this situation at all. C++ 4.2 always detects the situation, and for classes, where the usage could be disastrous, reports an error. For `structs` that use no member functions or other C++ features, the compiler gives a warning.

Solution: Change one of the names.

Error: Cannot assign int(*)(...) to int(*)(int, char).

C++ 3.0 has a bug that allows the assignment in the following program fragment:

```
int (*pfp)(int, char);
extern int foo(int, char);
void func() {
    pfp = (int (*)(...))foo;       // error
//  pfp = (int (*)(int, char))foo;    this works
}
```

This error is not detected in a direct cast as the right hand side of an assignment, and pointers of type `int(*)(...)` cannot be assigned to other pointers to functions. This bug does not exist in C++ 4.2.

Other Errors

Error: Cannot return int from a function which should return char*

The expression `(`*anything*, `0)` is not a null-pointer constant. For example, the following program results in a compilation error:

```
int error();
char * foo() {
    return (error(), 0); // error
    // should be:
    // return (error(),(char*) 0);
    // or:
    // error();
    // return 0;
}
```

Solution: Cast zero to the appropriate pointer type, or to pull the constant zero out of the comma expression.

Error: Cannot use {} to initialize <class name>.

According to the ARM, it is illegal to initialize a class with a base class using the aggregate initialization syntax. C++ 3.0 used to allow this as long as there were no virtual functions and no constructors.

Solution: Write a constructor and initialize the class with that.

Warnings

Warnings reflect changes in the language or errors undetected by C++ 3.0. A few warnings reflect areas where the current language definition makes code illegal, but we believe this definition should be changed.

Warning: AAAA is not accessible from BBBB

There are several contexts in which this message can appear:

- When returning a value of class type. The copy constructor must be accessible even if the compiler eliminates the use of the constructor. Thus the compiler may warn that `C::C (const C&)` is not accessible.

  Solution: Make the constructor `public`.

- When initializing a value of class type using the syntax `C cv =`*expression*. The situation is as described in the previous case.
- When using a `private` type name. C++ 3.0 frequently does not check access for type names; C++ 4.2 does.

  Solution: Correct the access of the type or remove the reference.

- When allocating a class on the heap using “new”, the compiler will generate code to delete the storage when an exception is thrown. To do this requires access to the appropriate “`operator delete()`”. Since some compilers do not check this access, and there is library code that depends on not checking this access, C++ 4.2 issues a warning instead of an error.
- Though the ARM and the DWP declare that defining a member of a `private` nested class outside the class is an error, there is doubt whether it will continue to be in the future.

Solution: Evaluate whether the type itself should be `private`. If so, you should see no other warnings about access to the type, and this warning can be ignored. Otherwise, make the type (though not necessarily the members) `public`. For example:

```
class A {
    class B {
                f();
    };
};
A::B::f()        // warning here
    { //stuff
    }
```

- C++ 3.0.1 allows the use of a `private` enumerator as a subscript in the definition of static arrays. For example

```
class Foo {
    enum {Max=27};
    static int b[Max];
};

int Foo::b[Foo::Max];
```

C++ 4.2 and the ARM do not allow this.

Solution: Make the enumerator `public`.

- C++ 3.0 allows the case where a derived class has a static object of its base class and the base class constructor is private or protected. C++ 4.2 and the ARM do not allow this. The following example illustrates this

```
class Base {
public:
    Base(const Base &);
    ~Base();

protected:
    Base(int);
};

class Derived : public Base {
public:
    Derived(int i): Base (i) {}
    static const Base xx;
};

const Base Derived::xx(1);
```

Solutions:

1. Make `Base::Base(int)` public, or

2. Add ‘`friend class Derived;`’ to the definition of `class Base`.

Warning: Default parameters are not allowed for AAAA

There are two cases that can cause this warning. The first is putting default parameters on overloaded operators. Such defaults are made illegal by the DWP, though C++ 3.0 allows them.

Solution: Remove the parameters from the operators. Since the only way the default parameters can be used is in an explicit call, this solution should cause no problem.

The second context is on pointers to functions. The ARM states that default parameters only apply to function declarations, not to pointers to functions. C++ 3.0 allows default parameters on such pointers, but its handling of them is inconsistent.

Solution: Remove defaults from function pointer declarations and update any calls that use them.

Warning: Formal argument AAAA of type BBBB has an inaccessible copy constructor

or

Warning: Formal argument AAAA of type BBBB in call to CCCC has an inaccessible copy constructor

These messages indicate that the object being passed as an argument cannot be copied, even though the compiler is able to eliminate the copy as an optimization. C++ 3.0 does not diagnose such errors, mostly because the language was not clarified on the situation until fairly recently.

Solution: Though the obvious solution is to make the copy constructor accessible, this may indicate an improper use of a class that was never intended to be copied. Check the program logic to determine the proper correction.

Warning: main() must have a return type of int

Though `main()` has always been required to return `int`, C++ 3.0 does not enforce this. Programs that return some other type may produce unpredictable results.

Solution: Change the definition of `main()` so it returns `int`.

Warning: The copy constructor for AAAA should take const AAAA&

or

Warning: The copy constructor for argument AAAA of type BBBB should take const BBBB&

or

Warning: The copy constructor for argument AAAA of type BBBB in call to CCCC should take const BBBB&

These warnings occur when the compiler has eliminated a copy constructor that would be illegal if called. C++ 3.0 detects this error when the copy constructor was actually used, but not when it was eliminated.

Solution: Since copy constructors that do not take `const` parameters are seldom desirable, the preferred solution is to modify the copy constructor.

Warning: Trailing comma in a parameter list

C++ 3.0 does not detect an erroneous trailing comma in an actual parameter list. For example, it allows

```
extern void f(int, int);
f(1, 2,)
```

Solution: Remove the extra comma.

Warning: Temporary created for argument AAAA

or

Warning: Temporary created for argument AAAA in call to BBBB

These warnings indicate the compiler created a temporary value for an argument that was a reference to a non-`const` type. For example:

```
extern void f(int&);
f(1);// called with a non-lvalue
```

This is always an error, but was accepted by earlier versions of C++. Even C++ 3.0 missed some cases, so this is a warning rather than a error.

Solution: The problem may be as simple as an inadvertently omitted `const` in the declaration, or there may be a program logic error.

Warning: Use of count in delete []

Earlier versions required an element count in the brackets of `delete[]`. C++ 4.2 allows this count, but ignores it.

Solution: Remove the count.

Type Warnings

Warning: AAAA was previously declared extern, not static

This is an error, and in most cases C++ 3.0 diagnosed this error correctly. The DWP states that any user-defined global version of `operator new()` or `operator delete()` is used by the library as well as the user's code. The functions must be global, so static versions are not allowed. This rule does not apply to placement versions of `operator new()`, only to the default version.

Warning: Assigning AAAA to the enum BBBB is obsolete

This anachronism is carried over from C++ 3.0. It is illegal to assign any value to an enumeration variable that is not of the enumerated type.

Solution: Change your program logic or cast the value to the enumerated type.

Warning: Attempt to redefine AAAA without using #undef

Macros cannot be redefined without an intervening `#undef`.

Solution: Insert `#undef AAAA` before redefining it.

Warning: Initialization without a class name is now obsolete

This warning is carried over directly from C++ 3.0. It occurs when writing a constructor for a derived class

```
class B {
    B(int);
};
class D: public B {
    D(int);
};
D::D(int i) : (i) {}
//            ^warning here
{}
```

Solution: Name the base class directly in the constructor initializer, as follows;

```
D::D(int i) : B(i) {}
```

Warning: Cannot delete a pointer to a constant (AAAA)

C++ 3.0 does not detect this error. C++ 4.2 does, but gives you a warning.

Solution: Do not delete a pointer to a constant, as it is almost certainly an error.

Warning: Empty declaration (probably an extra semicolon)

Empty declarations are not allowed, though empty statements are. This is usually the result of an editing error or an incorrect macro invocation. C++ 3.0 does not detect such empty declarations, so this is only a warning.

Warning: Objects of type AAAA must be initialized

This warning is produced when you use `new()` to allocate a `const` object without providing an initial value or a default constructor, as shown in the following example:

```
const int * ip = new const int;
```

This is an error, since constants of unknown value are not useful. The solution is to add an initializer. The preceding example would then become

```
const int * ip = new const int(0);
```

Warning: Temporary used for non-const reference, now obsolete

Earlier versions of C++ allowed the initialization of a non-`const` reference with an incompatible type or non-lvalue. This is obsolete now, but causes a warning message. To avoid the warning, check your program logic, and if you want a temporary, make an explicit one. For example, convert

```
short sv;
int & ir = sv;
```

to

```
short sv;
short & ir = sv;
```

or

```
int irtemp = sv;
int & ir = irtemp;
```

Formal argument AAAA of type BBBB is being passed CCCC

or

Formal argument AAAA of type BBBB in call to CCCC is being passed DDDD

These warnings indicate that the conversion required to pass the argument is illegal, but that C++ 3.0 did not detect the illegality. Failure to handle `const` and `volatile` properly on pointers is a major cause of this problem. You can usually correct it with an explicit cast, but the error is probably the result of a logic error. Passing a value of type `char**` to a parameter of type `const char**` is illegal in both C++ and ANSI C. This is not an error or oversight in the standard as such assignments open a hole in the type system and violate `const` safety. It is legal in C++ to pass a value of type `const char**` to a parameter of type `const char* const*`.

Other Warnings

Warning: Undefined character escape sequence

The only string or character constant escape sequences which are allowed in C++ are:

Table A-1 Escape Sequences

newline	`NL(LF)`	`\n`
horizontal tab	`HT`	`\t`
vertical tab	`VT`	`\v`
backspace	`BS`	`\b`
carriage return	`CR`	`\r`
form feed	`FF`	`\f`
alert	`BEL`	`\a`
backslash	`\`	`\\`

Table A-1 Escape Sequences *(Continued)*

question mark	`?`	`\?`
single quote	`'`	`\'`
double quote	`"`	`\"`
octal number	`ooo`	`\ooo`
hex number	`hhh`	`\xhhh`

The effect of any other escape sequence is undefined. C++ 4.2, like C++ 3.0, has replaced the entire escape sequence with the character following the backslash, which may or may not be what was intended.

Solution: Determine the intention of the code and substitute the correct character, possibly using a hex escape sequence.

Warning: Use AAAA:: for access to BBBB

Access to a nested type without an appropriate qualifier is allowed for compatibility with earlier versions of C++. Each such reference is flagged with this warning.

Solution: Use explicit qualification when using the nested type.

Warning: Using AAAA to initialize BBBB

C++ 3.0 was lax in its type checking, particularly when pointers to `const` were involved. Passing a value of type `char**` to a parameter of type `const char**` is illegal in both C++ and ANSI C. This is not an error or oversight in the standard as such assignments open a hole in the type system and violate `const` safety. It is legal in C++ to pass a value of type `const char**` to a parameter of type `const char* const*`.

Solution: Failure to handle `const` and `volatile` properly on pointers is a major cause of this problem. You can usually correct it with an explicit cast, but the error is probably the result of a logic error.

Warning: A declaration does not specify a tag or an identifier

ANSI C introduced the restriction that a declaration must define at least a tag or an identifier. The ARM is less precise, stating that a declaration introduces one or more names into a program. A declaration such as `extern int;` is useless and therefore flagged as an anachronism.

Solution: Eliminate the extraneous code.

Other Differences

Other differences between C++ 3.0 and C++ 4.2 follow.

`operator=()`

C++ 3.0 accepts any `operator=()` taking any arguments as a legitimate assignment operator for a class. C++ 4.2 and the DWP accept declarations of the form `X::operator=(X&)`.

Initializing Non-Aggregate Classes

C++ 3.0 allows the use of initializer lists with non-aggregate classes. This use is not allowed by the ARM (see the ARM, Section 8.4.1). C++ 4.2 enforces this rule. For example, the following is illegal:

```
class base {
    public:
    const char *name;
    unsigned int i1;
    unsigned int i2;
    unsigned int i3;
};

class foo : public base {
    public:
    char c;
    unsigned int i4;
};

static foo array[] = {
    {"hello world", 0, 0, 0, 'c', 0}
};
```

Using the Same Names in Base and Derived Classes

C++ 3.0 allows an enumerator name defined in a base class to be used as a derived class name. The following test case compiles without errors with C++ 3.0, but not with C++ 4.2

```
class foo {
public:
   typedef enum {
     bar,
     foobar
   } footype;
};

class bar  :  public foo {
public:
  bar();
  ~bar();
  bar(const bar &b);         // Error here
  bar &operator=(const bar &b); // Error here
};
```

You can disambiguate the name ‘bar’ in the derived class by using the keyword `class`, as follows

```
class foo {
public:
  typedef enum {
    bar,
    foobar
  } footype;
};
class bar : public foo {
public:
  bar();
  ~bar();
  bar(const class bar &b);
  class bar &operator=(const class bar &b);
};
```

Templates

If you are using templates with C++ 3.0, you may notice some differences when you move to C++ 4.2. C++ 3.0 generates (or instantiates) template functions when the program is being linked. A separate processor examines the object and library files, determines which template functions are needed, then uses the compiler to instantiate them. C++ 4.2 instantiates at compile time, generating those template instantiations needed by the module being compiled.

Link Order When Using Templates

The 4.0.1 compiler had an error in constructing the link line when templates were involved. In these cases, the old compiler would automatically place archives at the end of the supplied (and inserted template) object files. This could have unknown consequences when link order was important. The new compiler leaves the link order exactly as supplied, and only inserts template object files before the first non-object file (such as archives or shared library).

Specialization Registration

A *specialization* of a template is a user-defined version of a template that is normally generated by the compilation system. For example, for the template function:

```
template <class T> T foo ( T data );
```

A specialized version is:

```
int foo ( int data );
```

The compiler cannot differentiate between the declaration of a normal function and a template function. As a result, a special file, the options file, is available so that these specializations can be registered with the compiler. This options file resides in the template database, and contains many options that can determine how templates are generated. The options used to register specializations look like this for the above code fragment:

```
special foo(int);
```

The information needed to properly handle specializations of templates is covered in depth in the *C++ User's Guide*.

Template Repositories

Generated templates and their supporting files are stored in a directory, the template database. Except for the options file, all the files in the database are object files and state information files, maintained by the compiler. The difference between using the template database in C++ 4.2 versus C++ 3.0 is: when you use the `-ptr` option to specify the template database path, the path includes the database, named `Templates.DB`. For example:

In C++ 3.0:
    `-ptrMyDatabase`

corresponds to a database located in `./MyDatabase.`

In C++ 4.2:
    `-ptrMyDatabase`

corresponds to a database located in `./MyDatabase/Templates.DB`.

Central Database

You can use a central database to store generated templates. Do this by using the `-ptr` command-line directive to specify the location of your database. The inherent problem with a central database is that dependencies, types, and environmental data can change between executables sharing a common template and template database. Changes can lead to confusion, strange behavior at runtime, and missing data in the database. Template objects and dependencies in the relevant state-information files are based on template and type names. Because of potential conflicts between targets and violations of the one-definition rule, using templates across multiple targets may lead to unexpected behavior.

Multiple Databases

The template sources used during instantiation must be locatable from the directory where the referencing module is compiled. Use multiple databases by specifying multiple `-ptr` options on the command line. The first `-ptr` option is the writable template database. Whenever multiple databases are involved, specify `-ptr` as the first `-ptr` option. See the *C++ User's Guide* for details.

It is recommended that you not use multiple `-ptr` options on the command line when using templates. Although this should work with the current version of the compiler, it is not guaranteed to work in future releases.

Placing Objects in Shared Memory

Placing objects with virtual functions into shared memory requires that the virtual tables needed by those objects be at the same virtual address in all tasks that refer to the objects. To achieve this, control virtual table generation using the `+e0` and `+e1` command-line flags. `+e0` prevents generation of virtual tables. `+e1` generates virtual tables for every class defined in that compilation unit. Follow these steps:

1. Create a file containing only class definitions.
2. Compile the file with `+e1`, generating an object file containing only virtual tables.
3. Place the object file at the appropriate spot in virtual memory.

In C++ 4.2, the exception-handling code generates some internal class objects, so the compilation must include `-noex` as well as `+e1.`

For example, assume that you have header files `type1.h` and `type2.h`, and that all classes used within the program are defined in one of those files. You can compile most of the program using:

```
CC -c +e0 a.cc b.cc ...
```

and create a file `vt.cc` containing:

```
  #include "type1.h"
  #include "type2.h"
```

and compile it with

```
CC -c +e1 -noex vt.cc
```

The file `vt.o` contains the virtual tables needed by the program.

If you want to generate the virtual tables for the exception data structures, omit the `-noex` option on the `CC` command line.

Code Samples B

Here is an example C++ program illustrating C++ features. It consists of:

- A sample implementation of a string class
- A small program to test it

The string class is an example that, although not full-featured, illustrates features that a real class must have. Code Example B-1 is the header file `str.h` for the string class `string`.

Code Example B-1 Header File `str.h`

```
// header file str.h for toy C++ strings package
#include <string.h> // for C library string functions
class ostream;  // so we can declare output of strings

class string {
public:
    string();
    string(char *);
    void append(char *);
    const char* str() const;
    string operator+(const string&) const;
    const string& operator=(const string&);
    const string& operator=(const char*);
    friend ostream& operator<<(ostream&, const string&);
    friend istream& operator>>(istream&, string&);

private:
    char *data;
```

Code Example B-1 Header File `str.h` *(Continued)*

```
    size_t size;
};
inline string::string() { size = 0; data = NULL; }
inline const char* string::str() const { return data; }

ostream& operator<<(ostream&, const string&);
istream& operator>>(istream&, string&);
```

Code Example B-2 is an implementation file `str.cc` of the string class functions.

Code Example B-2 String Class File `str.cc`

```
//******************* str.cc *******************
// implementation for toy C++ strings package

#include <iostream.h>
#include "str.h"

string::string(char *aStr)
{
    if (aStr == NULL)
        size = 0;
    else
        size = strlen(aStr);

    if (size == 0)
        data = NULL;
    else {
        data = new char [size+1];
        strcpy(data, aStr);
    }
}

void string::append(char *app)
{
    size_t appsize = app ? strlen(app) : 0;
    char *holder = new char [size + appsize + 1];

    if (size)
        strcpy(holder, data);
    else
        holder[0] = 0;
```

Code Example B-2 String Class File `str.cc` *(Continued)*

```
//******************* str.cc *******************
    if (app) {
        strcpy(&holder[size], app);
        size += appsize;
    }
    delete [] data;
    data = holder;
}

string string::operator+(const string& second) const
{
    string temp;
    temp.data = new char[size + second.size + 1];

    if (size)
        strcpy(temp.data, data);
    else
        temp.data[0] = 0;

    if (second.size)
        strcpy(&temp.data[size], second.data);
    temp.size = size + second.size;
    return temp;
}

const string& string::operator=(const string& second)
{
    if (this == &second)
        return *this;   // in case string = string

    delete [] data;

    if (second.size)  {
        data = new char[second.size+1];
        size = second.size;
        strcpy(data, second.data);
    }
    else {
        data = NULL;
        size = 0;
    }
    return *this;
}
```

Code Example B-2 String Class File `str.cc` *(Continued)*

```
//******************* str.cc *******************
const string& string::operator=(const char* str)
{
    delete [] data;

    if (str && str[0])  {
        size = strlen(str);
        data = new char[size+1];
        strcpy(data, str);
    }
    else {
        data = NULL;
        size = 0;
    }
    return *this;
}

ostream& operator<< (ostream& ostr, const string& output)
{
    return ostr << output.data;
}

istream& operator>> (istream& istr, string& input)
{
    const int maxline = 256;
    char holder[maxline];
    istr.get(holder, maxline, '\n');
    input = holder;
    return istr;
}
```

Localization Support C

Support for languages other than English is described in this Appendix.

Native Language Support

This version of C++ supports the development of applications using languages other than English, including most European languages. As a result, you can switch the development of applications from one native language to another.

This C++ compiler implements internationalization as follows:

- It recognizes 8-bit characters from European keyboards supported by Sun.
- It is 8-bit clean and allows the printing of your own messages in the native language.
- It allows native language characters in comments, strings, and data.
- It allows you to localize the compile-time error messages files.

Locale

You can enable changing your application from one native language to another by setting the locale. Doing so changes some aspects of displays, such as date and time formats.

For information on this and other native language support features, read Chapter 6, “Native Language Application Support,” of the *System Services Overview* for Solaris software.

Compile-Time Error Messages

The compile-time error messages are on files called source catalogs so you can edit them. You may decide to translate them to a local language such as French or Italian. Usually, a third party does the translating. Then you can make the results available to all local users of `C++`. Each user of `C++` can choose to use these files or not.

Localizing and Installing the Files

Usually a system administrator does the installation. It is generally done only once per language for the whole system, rather than once for each user of `C++`. The results are available to all users of `C++` on the system.

1. **Find the message text files.**
   The file names are:

   - `SUNW_SPRO_SC_ccfe.msg`
   - `SUNW_SPRO_SC_libcomplex.msg`
   - `SUNW_SPRO_SC_libtask.msg`
   - `SUNW_SPRO_SC_driver.msg`

2. **Edit the message text files.**

   a. **Make backup copies of the files.**

   b. **In whatever editor you are comfortable with, edit the files.**
   The editor can be `vi`, `emacs`, `textedit`, and so forth.

   Preserve any existing format directives, such as `%1`, `%2`, `%3`, and so forth.

   c. **Save the files.**

3. **Generate the message database catalogs from the message text files.**
   The compiler uses only the formatted message database catalogs. Run the `gencat` program to create the database files.

a. **Generate the** SUNW_SPRO_SC_ccfe.cat **message database from the** SUNW_SPRO_SC_ccfe.msg **text file using** gencat:

```
demo% gencat SUNW_SPRO_SC_ccfe.cat \
                                  SUNW_SPRO_SC_ccfe.msg
```

Do this for each changed .msg file.

4. **Make the message database catalogs available to the general user.**
Either put the catalogs into the standard location or put the path for them into the environment variable NLSPATH.

   a. **Put the catalogs in the standard location.**
   Put the files into the directory indicated:

   /opt/SUNWspro/lib/locale/*lang*/LC_MESSAGES/

   where *lang* is the directory for the particular (natural) language. For example, the value of *lang* for Italian is it.

   b. **Set up the environment variable.**
   Put the path for the new files into the NLSPATH environment variable. For example, if your files are in /usr/local/MyMessDir/, then use the following commands.

   In a sh shell:

```
demo$ NLSPATH=/usr/local/MyMessDir/%N.cat
demo$ export NLSPATH
```

   In a csh shell:

```
demo% setenv NLSPATH  /usr/local/MyMessDir/%N.cat
```

   The NLSPATH variable is standard for the X/Open environment. For more information, read the X/Open documents. See also gencat(1) and catgets(3C) for more information on message catalogs.

The CC driver sets up the environment variable NLSPATH before invoking the compiler ccfe. The existing value of NLSPATH, if any, is prepended to the standard location. In the above example, before invoking ccfe, the CC driver will set NLSPATH to:

```
/usr/local/MyMessDir/%N.cat:   \
/opt/SUNWspro/lib/locale/%L/LC_MESSAGES/%N.cat
```

Using the File After Installation

You use the file by setting the environment variable LC_MESSAGES. This setup is generally done once for each developer.

Example: Set the environment variable LC_MESSAGES, assuming standard install locations, and the messages are localized in Italian:

In a sh shell:

```
demo$ LC_MESSAGES=it
demo$ export LC_MESSAGES
```

In a csh shell:

```
demo% setenv  LC_MESSAGES it
```

If you want to use the localized messages for libcomplex and libtask, then you must set the environment variable NLSPATH before executing your program. You should do this even if you have installed the localized catalogs in the standard location.

Index