C++ Programmer's Guide

Document Number 007-0704-110

CONTRIBUTORS

Written by Douglas B. O'Morain and Renate Kempf
Illustrated by Douglas B. O'Morain
Production by Ruth Christian
Engineering contributions by Trevor Bechtel, Ashok Chandramouli, T.K. Lakshman, Michey Mehta, C. Murthy, and John Wilkinson
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

© Copyright 1995, 1996 Silicon Graphics, Inc.— All Rights Reserved
The contents of this document may not be copied or duplicated in any form, in whole or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND
Use, duplication, or disclosure of the technical data contained in this document by the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013 and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR Supplement. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd., Mountain View, CA 94043-1389.

Silicon Graphics and IRIS are registered trademarks and IRIX, IRIS IM, IRIS ViewKit, Graphics Library™, Indigo Magic, Indigo Magic Desktop, CASEVision, CASEVision/WorkShop, and CASEVision/WorkShop Pro C++ MIPSpro™ are trademarks of Silicon Graphics, Inc. Open Software Foundation, Motif, OSF, OSF/Motif are trademarks of the Open Software Foundation, Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd. Borland C++ is a registered trademark of Borland International, Inc.

C++ Programmer's Guide
Document Number 007-0704-110

Contents

List of Examples vii

List of Figures ix

List of Tables xi

About This Guide xiii
What This Guide Contains xiii
What You Should Know Before Reading This Guide xiii
Related Information xiv
The Standard Template Library xiv
Conventions Used in This Guide xv

1. **Understanding the Silicon Graphics C++ Environment** 1
Silicon Graphics C++ Environment 1
New Features and ABI Changes in the 64-bit Compiler 2
    Operators new[] and delete[] 3
    Built-in bool Type 3
    Built-in wchar_t Type 4
    Assignment to ‘this’ 5
    Exception Handling 5
    Runtime Type Identification 7
    Other ABI Changes 8
Using the Compilers 9
    64- Versus 32-Bit Compilation 10
    CC Command Line 11
    Sample Command Lines 11
cfront Compatibility 12
C++ Libraries 12
Debugging 12

2. Compiling, Linking, and Running C++ Programs 13
Compiling and Linking 13
Translators and Drivers 13
Compilation 14
Multi-Language Programs 17
Translator Options 17
Object File Tools 19

3. C++ Dialect Support 21
About the Front End 22
New Language Features 22
Non-implemented Language Features 24
Anachronisms Accepted 25
Extensions Accepted in Default Mode 26
Extensions Accepted in Cfront Compatibility Mode 27
Cfront Compatibility Restrictions 32

4. Common Pitfalls 33
Problems Involving C Linkage 33
Problems With Order of Specification of Libraries 34

5. Using Templates 37
Template Instantiation 37
Automatic Instantiation 38
Meeting Instantiation Requirements 38
Automatic Instantiation Method 39
Details of Automatic Instantiation 39
Implicit Inclusion 41
Explicit Instantiation 41
Command Line Options for Template Instantiation 42
Command Line Instantiation Examples 44
Pragmas for Template Instantiation 46
Specialization 48
Building Shared Libraries and Archives 48
Limitations 48

How to Transition From cfront 50
- Mapping Template Options From cfront to CC 50
- What to Do If You Use Object Files From cfront's Repository 52
- What to Do If You Use Multiple Repositories 52

Template Language Support 53

A. C and C++ Pragma Directives 57

Glossary 61

Index 63

List of Examples

Example 1-1 Exception Handling 6
Example 1-2 Runtime Type Identification (RTTI) 8

List of Figures

Figure 1-1 Silicon Graphics C++ Environment 2

Figure 2-1 The Compilation Process 15

List of Tables

Table 1-1 32- and 64-bit Silicon Graphics Systems 9

About This Guide

This guide describes how to use the Silicon Graphics® C++ compiler environment. It discusses the two native C++ compilers for producing 32- and n32/64-bit objects, respectively.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1, "Understanding the Silicon Graphics C++ Environment," describes the Silicon Graphics C++ environment and the issue of *cfront* compatibility.
- Chapter 2, "Compiling, Linking, and Running C++ Programs," describes how to compile, link, and run C++ programs in the Silicon Graphics C++ environment.
- Chapter 3, "C++ Dialect Support," describes the C++ language supported by the Silicon Graphics C++ compilers.
- Chapter 4, "Common Pitfalls," discusses some common problems with C++ libraries and how to diagnose and solve them.
- Chapter 5, "Using Templates," discusses how C++ templates are used in the Silicon Graphics C++ environment.
- Appendix A, "C and C++ Pragma Directives," discusses the C and C++ pragmas available with the compilers.

The glossary defines key terms for the Silicon Graphics C++ environment.

What You Should Know Before Reading This Guide

This guide assumes that you are familiar with C, C++, object-oriented programming, shared libraries, and dynamic loading.

Related Information

The following manuals provide reference information about the Silicon Graphics implementation of the C++ language. Note that these manuals describe the *cfront* compiler, and parts of the description of the C++ language are now obsolete.

- *C++ Language System Overview* contains an overview of newer language features of C++. Most of the extensions take the form of removing restrictions on what can be expressed in C++.
- *C++ Language System Product Reference Manual* contains a general description of the C++ language.
- *C++ Language System Library* discusses the stream support in the C++ library and describes a data-type complex that provides the basic facilities for using complex arithmetic in C++.

The following manual provides related information that you may need when using the Silicon Graphics C++ environment.

- *MIPSpro Compiling and Performance Tuning* discusses how to compile, and tune the performance of programs written in the Silicon Graphics development environment (C, Fortran, and C++).
- *dbx User's Guide* discusses how to debug your code in the Silicon Graphics development environment.
- The *C Language Reference Manual* contains information about C/C++ multiprocessing compiler directives in Chapter 11.

The Standard Template Library

The Standard Template Library (STL), is an extensible generic library of C++ algorithms and data structures. It is part of the emerging ANSI/ISO C++ standard. The library is shipped together with the C++ compiler. Documentation in html format is available via the SGH home page (www.sgi.com).

Conventions Used in This Guide

These are the typographical and graphic conventions used in this guide:

- **Bold**—Functions, option flags, and classes
- *Italics*—Filenames, button names, field names, variables, emphasis, glossary terms, and IRIX commands
- Regular—Menu and window names, data types, keywords, and text
- "Quoted"—Menu choices
- `Fixed-width`—Code examples and command syntax
- **`Bold fixed-width`**—User input. Nonprinting **`<keys>`** are bracketed

Chapter 1

Understanding the Silicon Graphics C++ Environment

This chapter describes the Silicon Graphics C++ compiler environment and contains the following major sections:

- "Silicon Graphics C++ Environment" on page 1 discusses the different Silicon Graphics 7.0 C++ compilers for IRIX 6.x systems.
- "New Features and ABI Changes in the 64-bit Compiler" on page 2 describes the new features and ABI changes that are available in the 6.2 and 7.0 versions of the Silicon Graphics C++ compilers.
- "Using the Compilers" on page 9 discusses the differences between the 32- and 64-bit versions of the Silicon Graphics compilers, shows the command lines for the compilers, and gives some examples of typical command lines.
- "cfront Compatibility" on page 12 discusses the restrictions on C++ code that are enforced by the Silicon Graphics C++ environment, but were not enforced by *cfront*.
- "C++ Libraries" on page 12 discusses the C++ libraries in the Silicon Graphics C++ environment.
- "Debugging" on page 12 discusses the Silicon Graphics C++ debugging environment.

Silicon Graphics C++ Environment

The Silicon Graphics 7.0 C++ environment is available in two varieties, targeted for IRIX 6.x systems. See Figure 1-1 for details.

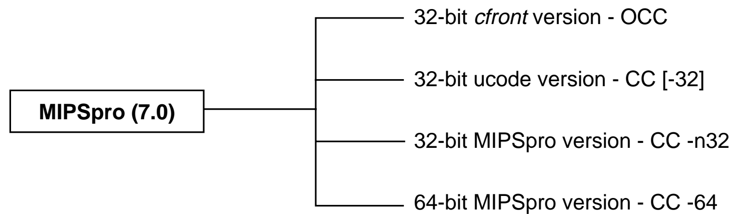

Figure 1-1 Silicon Graphics C++ Environment

As shown in Figure 1-1, there are 32- and a 64-bit versions of the C++ compiler for the IRIX 6.x operating system. For ease of migration, the old Silicon Graphics C++ compiler (*OCC*), based on *cfront*, is still available, although unsupported as of this release. The supported C++ compilers for the 6.x system are listed below:

CC	32-bit native ucode C++ compiler (the same as *CC -32*)
CC -n32	32-bit native MIPSpro compiler (includes improved optimization)
CC -64	64-bit native MIPSpro compiler
OCC	32-bit C++ compiler, based on C++ to C translation using *cfront*.

On 64-bit hardware, *CC* generates 64-bit code by default (without using the *-64* extension explicitly), while on 32-bit hardware, it generates 32-bit code by default. Note that you cannot mix object files compiled by different compilers.

New Features and ABI Changes in the 64-bit Compiler

Both the 64-bit and *-n32* compilers have some new features. Some of these features were first introduced in MipsPro 6.2, and the rest of these features were introduced in MipsPro 7.0. Exception Handling is available in the 32-bit compiler (*CC* -32), but the rest of these new features are not available in the 32-bit compiler; you'll have to protect the uses of these new features with an *#ifdef* if you want your code to be portable across all Silicon Graphics C++ compilers.

There is a new built in macro, **__EDG_ABI_COMPATIBILITY_VERSION**, which is undefined in the default 32-bit compiler, but set to the integer value `229` in the *-64* and *-n32* compilers. You can either use this macro to protect your use of new language features described in this section, or you can create your own *#ifdef*.

Operators new[] and delete[]

The 64-bit compiler now implements the array variants of operator **new** and **delete** from the most recent ANSI C++ drafts. All calls to allocate and deallocate arrays of objects using `new Classname[n]` and `delete[] Classname` will go through operators **new[]** and **delete[]** respectively, instead of operator **new** and operator **delete**.

This implies the following:

- If you override the global `::operator new()`, you probably don't have to do anything; the default library `::operator new[]()` simply turns around and calls `::operator new()` to allocate memory.

  (It is recommended that you also redefine `::operator new[]()` if you redefine `::operator new()`. The same also applies to operator **delete**.)

- If you define a placement operator **new**, such as,

  ```
  operator new(size_t, other parameters);
  ```

  You must define an additional operator **new**, such as,

  ```
  operator new[](size_t, other parameters)
  ```

  to be bound to calls of the form:

  ```
  new(other parameters) Classname[n];
  ```

  If you do not do this, the compiler issues an error about an appropriate operator **new** not being declared.

- The same applies for class-specific `operator new()` and `operator delete()`: you should also define a corresponding class-specific `operator new[]()` or `operator delete[]()` in each case.

Note: You will have to protect these declarations under a macro like the one described in the introduction to this section, or you will get compiler errors if you try to compile with the default 32-bit compiler.

Built-in bool Type

There is now a built-in **bool** type in the *-64* and *-n32* compilers (but not in the *-32* compiler). In addition, the keywords **true** and **false** are now keywords when **bool** is supported as a built-in type, with the obvious values (**bool** equivalents of 1 and 0 respectively).

To take advantage of this type in a fashion that is portable between *-32* and *-64/-n32*, you can declare a **bool** type for *-32* as follows:

```
#ifndef _BOOL
/* bool not predefined */
typedef unsigned char bool;
static const bool false = 0;
static const bool true = 1;
#endif /* _BOOL */
```

The macro **_BOOL** is pre-#defined to be `1` when the **bool** keyword is supported.

Note: The *-LANG:bool=off* option in the *-64/-n32* compiler can be used to disable built-in **bool**, **true** and **false** (in case you have already used any of these identifiers in your program), making it behave like the *-32* compiler in this regard.

Built-in wchar_t Type

The type **wchar_t** is a keyword and built-in type in the *-n32/-64* compilers. It is analogous to the **wchar_t** type defined in *stddef.h*, and in fact, this file can be safely included into an *-n32/-64* compile, and will not interfere with the built-in **wchar_t**. When the compiler supports **wchar_t,** it also defines a macro called **_WCHAR_T**; this allows you to write code that is portable across both the -32 and the -64/-n32 compilers.

For instance, you can now read and write **wchar_t**s directly (and they won't be read and written as **long**s, but will actually be read and written as multi-byte characters in the locale of the execution of the program).

Note: Since the built-in **wchar_t** is considered a distinct type (in other words, not a synonym for **int** or **long**), there is a potential for problems. For example, consider if you attempt to pass a built-in **wchar_t** to a function that is overloaded on other integral types, but not specifically on **wchar_t** as follows:

```
extern void foo(int);
extern void foo(long);

wchar_t w;

foo(w);// OK in -32, ERROR in -n32/-64
```

The fix for this is to declare a variant for **wchar_t**, but only when **__EDG_ABI_COMPATIBILITY_VERSION** >= 229:

```
extern void foo(int);
```

```
extern void foo(long);
#if __EDG_ABI_COMPATIBILITY_VERSION >= 229
extern void foo(wchar_t);
#endif
```

If you do not *#ifdef* it this way, then the *-32* compiler will complain that `foo(long)` has already been declared (since in -32, **wchar_t** is a synonym for **long**).

The option *-LANG:wchar_t=off* can be used to disable recognition of the **wchar_t** keyword.

Assignment to ‘this’

You can no longer assign to **this** in a constructor to implement *cfront* v1.2-style class-specific allocation. The allocator is always called directly by the caller before entering the constructors.

Thus, constructors generated by the *-n32/-64* compiler will never call operator **new()** implicitly if you pass in a NULL pointer as the first argument.

The calls to operator **delete()**, however, are still embedded in the destructors themselves, because this is necessary in order to support the two-argument form of operator **delete()**.

Exception Handling

Exception handling is supported in the Silicon Graphics 7.0 compilers by default in the n32/64-bit mode; it can be turned of by using the *-LANG:exceptions=off* option. It is also supported in the -32 compiler by using the *-exceptions* flag. The exception handling constructs of C++ permit the user to write code that detects an abnormal execution state of the program and take appropriate action. For instance, if a garbage collector runs out of memory to allocate, the routine may signal an exception which can be handled by displaying an appropriate message and possibly increasing the allocatable heap size.

The Silicon Graphics C++ compilers provides mechanisms for catching (handling) exceptions in a different scope than the scope where they were raised. The implementation of exceptions ensures that for programs that do not throw exceptions there is no appreciable performance penalty.

The following example illustrates the basic syntactic constructs used in exception handling: **try**, **throw**, and **catch**.

Example 1-1 Exception Handling

```
void allocator() {
........
if OutOfSpace()
 throw MemOverflowError();
........
}

main() {
.........
.........
try { // wrapper for code that may throw exceptions
mem * = allocator();
.........
}

catch (MemOverflowError) { // Exception Handler
 cout << "Exception during allocate";
.......

}
```

The allocator function raises an exception if it runs out of space. In the main program, the call to allocator is enclosed within a **try** block, a program region where exceptions may be **thrown**. If indeed an exception is raised in the call to allocator, then control shifts to the **catch** clause which catches the Memory Overflow error and does suitable error handling.

The syntax of a **try**/**catch** block combination is as follows:

```
try {
 .......
}
catch (int ) { // catch exceptions of one kind (int) thrown in the try
block above
....
}

catch(float ) {// catch exceptions of another kind (float) thrown in
the try block above
...
}
catch(...) {// catch ALL exceptions
....
}
```

A **try** block can be followed by zero or more **catch** blocks. If during the execution of a **try** block, no exceptions are raised then the control shifts to immediately after the last **catch** block. If on the other hand an object of a certain type is **thrown** (in other words, an exception is raised) during the execution of a **try** block, then the **catch** whose type specification matches the type of the **thrown** object is chosen and control transfers to that handler. The **catch** block `catch(...)` indicates that all exceptions are caught in this block.

Runtime Type Identification

This feature is only available in the 7.0 -n32/-64 compilers. C++ provides static type checking, which helps detect compile-time type errors. However, there are situations in which the type of an object may only be known at runtime and it becomes necessary to provide some form of type safety. Specifically, given an object of a base class, you may wish to determine whether in fact it is an object of a specific derived class of that base class. Consider the following example:

Example 1-2 Runtime Type Identification (RTTI)

```
class base {
public:
 virtual ~base() {};
}
Class derived : base {
public:
 void ctor() {};
}
```

You may wish to write a function that takes as argument a pointer to base, and calls **ctor** only if that pointer is in fact a pointer to **derived**. To do this you need a mechanism for determining whether a pointer to a given base class is in fact a pointer to a derived class. You can do this as follows using the **dynamic**-**cast** facility of C++:

```
f(base *pb) {
 if (derived *pd = dynamic_cast <derived *> (pb))
 pd -> ctor();
.....
}
```

The **dynamic_cast** <..> operation verifies that the pointer **pb** is actually pointing to an object of derived class rather than of base class, otherwise dynamic_cast returns 0.

In addition to **dynamic_cast**, C++ also provides an operator **typeid**() which determines the exact type of an object. This operator returns a reference to a structure **typeinfo** which represents the type name of its argument.

Other ABI Changes

- The object layout has been modified somewhat between -32 and *-n32*, specifically for classes that have virtual base classes inherited along more than one path. If you have a dependency on the exact layout of such objects, this may cause you some difficulties.
- The name mangling is different for *-n32/-64* compilers; it is almost exactly the same as for *-32*, except that the function designator *F* (as in `foo__FUi`) is *G* in the *-n32/-64* mangled names (for example, `foo__GUi`).

  If you have calls to mangled C++ names in your assembly, C or Fortran code, or you do dynamic name lookups using **dlsym** on mangled C++ names, you will have to take this into account.

- Virtual function tables (internal to the implementation) have been expanded in *-n32/-64*, so that you can now actually have subobjects that are larger than 32KB, and more than 32K virtual functions.

  The limits are now 4GB for subobject sizes (both in *-n32* and *-64*), and 4 billion virtual functions (though we doubt you'd actually get anywhere near the latter limit). Even the subobject size is an issue only if a class larger than 4GB is a non-rightmost base class in a multiple-inheritance situation; for single-inheritance, the base class sizes should not be limiting.

Using the Compilers

This section discusses how to use the Silicon Graphics compilers to compile your C++ programs. It describes the differences between the 64- and 32-bit versions of the compiler, describes the *CC* and *OCC* command lines (and some of the more commonly used options), and contains some examples.

The default compiler depends on your hardware: on 64-bit systems, *CC* defaults to *-64* mode; on 32-bit systems, *CC* defaults to *-32* mode. If you use *CC* with options supported by *OCC* but not supported by standard *CC*, or you use *CC* with the *-use_cfront* option, you invoke *OCC*. (For examples of 32- and 64-bit Silicon Graphics systems, see Table 1-1.)

Table 1-1 32- and 64-bit Silicon Graphics Systems

32-bit Systems	64-bit Systems
Indy	Challenge
Indigo	Power Challenge
Indigo 2	Onyx
Crimson	Power Onyx
	Power Indigo 2

Note: The *-use_cfront* option is ignored in -64 mode.

64- Versus 32-Bit Compilation

CC -64 and *CC -n32* are both native compilers that are based on the same front end, *fecc*. *fecc* has 64-bit pointers, addresses, and long ints for *CC -64*, and 32-bit pointers, addresses, and long ints for *CC -n32*. *CC -32* has a different front end, *edgcpfe*, with 32-bit pointers, addresses, and long ints.

The major difference between fecc and edgcpfe is code optimization—the code compiled by *fecc* is much more highly optimized than that generated by *edgcpfe*. Note that fecc may take longer to compile code than *edgcpfe* due to the increased optimization performed.

The default compilation mode for *CC -64* and *CC -n32* is *-mips4*. The default mode for *CC -32* is *-mips2*. To run *-64* and *-n32* executables on an R4x00 Silicon Graphics systems (every system listed in Table 1-1 except the Power systems), you need to explicitly specify *-mips3*.

Note: 64-bit objects are incompatible with 32-bit objects, and they cannot be linked together. 64-bit objects can only be created on 6.x-based systems. You can do this as follows:

- Specify the *-64* option on the IRIX 6.x command line to compile source files for 64-bit objects. This is the default for the MIPSpro compilers installed on 64-bit IRIX 6.x systems.
- Specify the *-32* option on the IRIX 6.x command line to compile source files for 32-bit objects. This is the default for the MIPSpro compilers installed on an 32-bit IRIX 6.x systems.

The compiler back-end (optimizer and code generator) is different in *-32* and *-64* modes.

The warning options used by the *-woff* option are different between *CC -64* and *CC -32*.

For information on the precompiled header mechanism, see *MIPSpro Compiling and Performance Tuning*.

Refer to *MIPSpro Compiling, Debugging, and Performance Tuning* for a more complete discussion on how to set up the IRIX environment for *-32* versus *-64* compilers. Refer to the *MIPSpro Porting and Transition Guide* for further information on *-64* compilers.

CC Command Line

The command line for *CC* is shown below.

```
CC [ option ] . . . file . . .
```

CC compiles with many of the same options as cc(1). *CC -64* is the default on 6.x (64-bit) systems, and *CC -32* is the default on 5.x (32-bit) systems.

Note: *cfront* compatibility mode is disabled by default when you compile in 64-bit mode.

See the *CC(1)* reference page for more information.

Sample Command Lines

Some typical C++ compiler command lines are given below.

- To suppress the loading phase of your compilation and compile only one program, the command line is the following:

  ```
  CC -c program
  ```

- To compile with full warning about questionable constructs, the command line is the following:

  ```
  CC -fullwarn program1 program2 . . .
  ```

- To compile with warning messages off, the command line is the following:

  ```
  CC -w program1 program2 . . .
  ```

- To compile in 64-bit mode with *cfront* compatibility enabled, the command line is the following:

  ```
  CC -64 -cfront program1 program2 . . .
  ```

- To compile in 32-bit mode with cfront compatibility disabled, the command line is the following:

  ```
  CC -32 +p program1 program2 . . .
  ```

cfront Compatibility

The Silicon Graphics compilers (with the exception of *OCC*) force you to adhere to C++ code standards more strictly than *cfront* does. Code that you compiled successfully with *cfront* may not compile under the Silicon Graphics C++ environment, even in *cfront* compatibility mode. You must compile with *OCC* to get exact *cfront* compatibility. Chapter 3, “C++ Dialect Support” discusses details of the *-cfront* option, which enables partial *cfront* compatibility.

C++ Libraries

By default, all C++ programs link with the standard library *libC.so*. This library contains all the *iostream* library functions, as well as the C++ storage allocation functions **::new** and **::delete**. All -n32/-64 programs will also link with the *libCsup.so library, which provides exception handling and Run-time Type Information support; this library is also used for -32 links if -exceptions has been specified.*

Silicon Graphics also provides the complex arithmetic library *libcomplex.a*. If you want to use this package you must explicitly link with this library. For example,

```
CC complexapp.c++ -lcomplex
```

See the *C++ Language System Library* for more information on the *complex* and *iostream* libraries.

Debugging

You can debug your C++ programs with the *dbx* or WorkShop debugger. For complete information on *dbx*, see the *dbx User's Guide*. For complete information on the WorkShop debugger, see the *Debugger User's Guide*.

Compiling, Linking, and Running C++ Programs

This chapter contains the following major sections:

- “Compiling and Linking” describes the compilation environment and how to compile and link C++ programs. Some examples show how to create separate linkable objects in C++, C, Fortran, or other languages, and how to link them into an executable program.
- “Translator Options” on page 17 describes the command line options that can be provided to the C++ translator.
- “Object File Tools” on page 19 briefly summarizes the capabilities of the tools that provide symbol and other information on object files.

Compiling and Linking

This section discusses Silicon Graphics C++ compiling and linking.

Translators and Drivers

Programs called *drivers* invoke the major components of the compiler system. Those components, their functions, and their place in the compilation process are discussed in the following sections. The *CC(1)* command invokes the driver that controls compilation of your C++ source files. The syntax is as follows:

`CC [`*options*`]` ***filename***`.C [`*options*`] [`***filename2***`.C ...]`

where:

CC invokes the various processing phases that translate, compile, optimize, assemble, and compile-time link the program.

options represents the driver options, which give instructions to the processing phases. Options can appear anywhere in the command line. The options interpreted by *CC* are discussed in “Translator Options” on page 17 in this chapter.

filename.C is the name of the file that contains the C++ source statements. The filename must end with one of the following acceptable suffixes: *.C, .c++, .c, .cc, .cpp, .CPP, .cxx* or *.CXX*.

Compilation

The compilation process shown in Figure 2-1 is that of the C++ source file *foo.C,* as it would be compiled by this command line:

```
CC -o foo foo.C
```

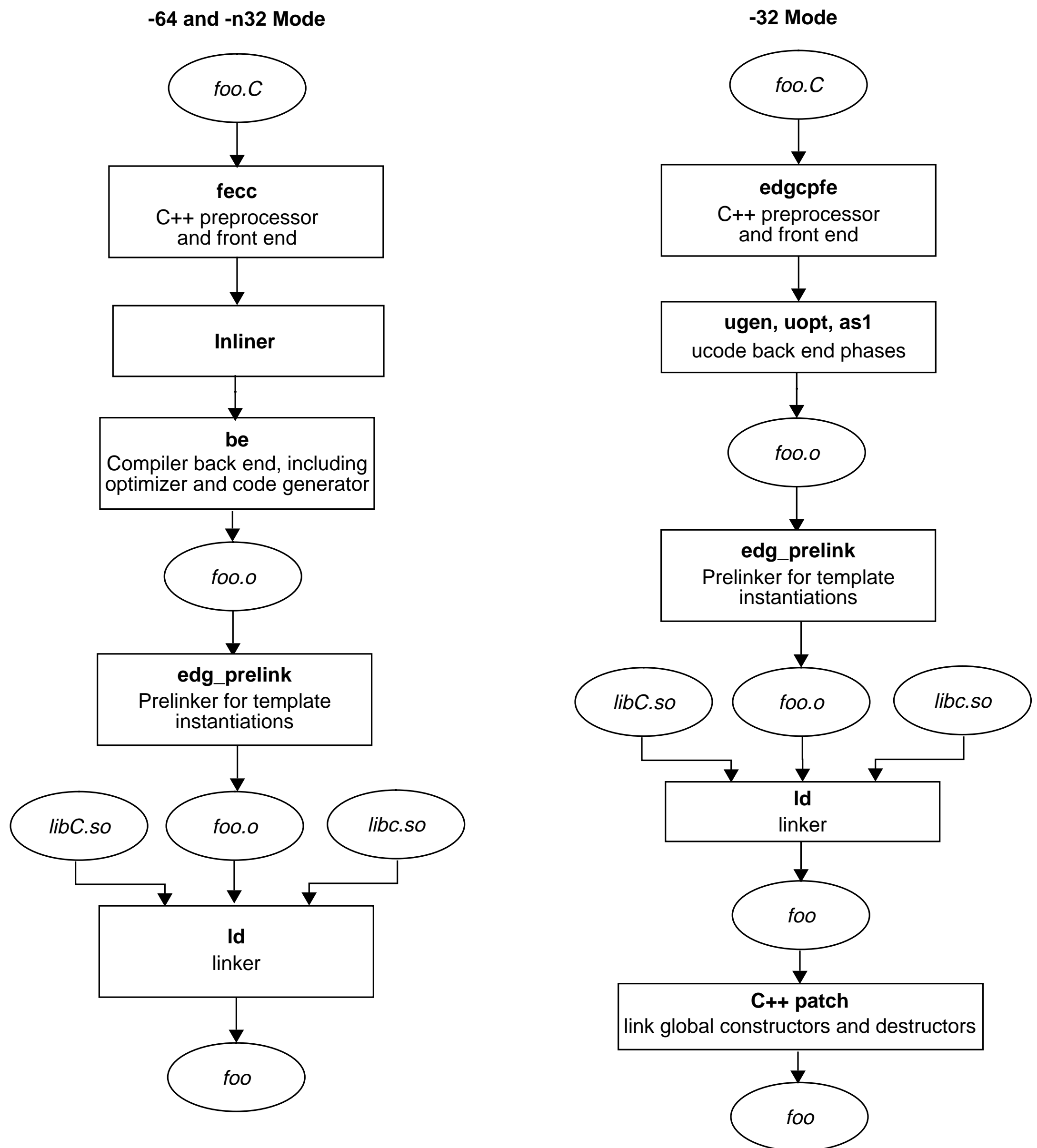

Figure 2-1 The Compilation Process

The following steps further describe the stages of compilation:

1. You invoke *CC* on the source file, which ends with the suffix *.C.* The other acceptable suffixes are *.C, .c++, .c, .cc, .cpp, .CPP, .cxx* or *.CXX.*
2. The source file then passes through the C++ preprocessor, which is built into the C++ front end (*fecc, edgpcfe*).
3. The complete source is then processed by the C++ front end (*fecc* or *edgcpfe*), which produces an intermediate representative from a syntactic and semantic analysis of the source.

   This stage may also produce a prelink (*.ii*) file, which contains information about template instantiations.
4. The back end (*be* in -n32/*-64* mode) generates optimized object code (*foo.o*).
5. If you want to stop the compilation at this phase, and produce object code suitable for later linking, use the following command:

   ```
   CC -c foo.c
   ```

   The object file *foo.o* is the result.
6. *edg_prelink* processes the *.ii* files associated with the objects that will be linked together. It then recompiles sources to force template instantiation.
7. The object files are sent to the linker *ld*(1), which links the standard C++ library *libC.so* and the standard C library *libc.so* to the object file *foo.o* and to any other object files that need to be linked to produce the executable.
8. In *-32* mode only, the executable object is sent to *c++patch,* which links it with global constructors and destructors. If global objects with constructors or destructors are present, the constructors need to be called at run time before function **main()** is called, and the destructors need to be called when the program exits. *c++patch* modifies the executable *(a.out)* to insure that these constructors and destructors get called.

Multi-Language Programs

C++ programs can be compiled and linked with programs written in other languages, such as C, Fortran, and Pascal. When your application has two or more source programs written in different languages, you should compile each program module separately with the appropriate driver. Then you can link them in a separate step. You can create objects suitable for linking by specifying the *–c* option. For example:

```
CC -c main.c++
f77 -c module1.f
cc -c module2.c
```

The various compilers would produce three object files: *main.o, module1.o,* and *module2.o.* Since the main module is written in C++, you should use the *CC* command to link. Except for C, you must explicitly specify the link libraries for the other languages with the *–l* options. For example, to link the C++ main module with the Fortran submodule, you would use the following command:

```
CC -o almostall main.o module1.o -lF77 -lI77 -lisam -lm
```

It is best to use CC to link if any object file is generated by C++.

For more information on C++ libraries, see “C++ Libraries” in Chapter 1.

Translator Options

This section contains a summary of the most important *CC* translator options. See the reference page for *CC*(1) for a complete description of all the options. See *ld*(1) for a description of the linker options, and *cc*(1) for a description of the options interpreted by the standard C compiler. See also the information in *MIPSpro Compiling, Debugging and Performance Tuning.*

–E Run only *cpp*(1) on the C++ source files and send the result to standard output. This option is useful, for example, if you want to see exactly which files were included in your compilation.

–c Produce object files only, suppressing the link phase.

–o output Name the final output file *output*. For example,

```
CC –o foo foo.C
```

produces an executable called *foo* instead of the default *a.out*. The command is shown below.

```
CC –c –o bar.o foo.C
```

produces an object file called *bar.o* instead of the default *foo.o.*

–n (*-32* mode), *-show0* (*-n32/-64* mode)
Print commands generated by *CC* but do not execute them.

-show (*-n32/-64* mode)
Print commands as they are executed. Short for *verbose* output.

+d (*-32* mode), *-noinline* (*-n32/-64* mode)
Do not attempt inline substitution for calls to functions declared as **inline**.

-fullwarn (*-n32/-64* mode)
Warn about all questionable constructs. Without the *+w* option, the translator issues warnings only about constructs that are almost certainly problems.

+p (*-32* mode), Disallow all anachronistic constructs. Ordinarily, the translator warns about anachronistic constructs. Under *+p* (for pure), the translator will not compile code containing anachronistic constructs, such as "assignment to this." See the *USL C++ Language System Product Reference* for a list of anachronisms.

In *-32* mode, *+p* also disables *cfront* compatibility mode, enforcing a stricter, more standard language definition. In *-64* mode, by default anachronisms are disallowed and the stricter definition is the default enforced.

-use_cfront (*-32* mode)
Use *OCC* instead of *CC*. Use with caution, since the use of this option is being deprecated.

-cfront (*-n32/-64* mode)
Compile in *cfront* compatibility mode. This is the default in *-32* mode. The *+pp* option will disable *cfront* compatibility mode.

-anach (*-n32/-64* mode)
Allows anachronisms in *-n32/-64* mode.

–nocpp Skip the preprocessing stage.

-*exceptions* (-*32* mode)
Enable exception handling constructs in the language. Code compiled with and without exceptions cannot generally be mixed. See the -*LANG* options for details. Note that code compiled with -*exceptions* in -*32* mode cannot be linked with code compiled with exception-handling on in the -*n32/-64* mode.

-LANG:...(-n32/-64 mode)
exceptions[=(ON | OFF)]: Enable exception handling constructs in the language. Code with and without exception handling cannot generally be mixed. Specifically, the scopes crossed between throwing and catching an exception must all have been compiled with *exceptions*=ON. Should be used with caution. (Default is ON.)

bool[=(ON | OFF)]: Enable the predefined bool data type, along with the predefined values TRUE and FALSE. Should be used with caution only to suppress this type in old code which defines **bool** itself. Because this option changes the mangling of function names with **bool** parameters, all files comprising a program should be compiled with consistent options. (Default is ON.)

wchar_t[=(ON | OFF)]: Enable the predefined **wchar_t** data type. Should be used with caution only to suppress this type in old code which defines **wchar_t** itself. Because this option changes the mangling of function names with **wchar_t** parameters, all files comprising a program should be compiled with consistent options. (Default is ON.)

Object File Tools

For information on the object file tools available to you, consult the *MIPS Compiling and Performance Tuning Guide.* The following tools are of special interest to the C++ programmer:

nm The *nm* tool can be used to print symbol table information for object files and archive files.

c++filt This C++-specific tool translates the internally coded (mangled) names generated by the C++ translator into names more easily recognized by the programmer. You can, for example, pipe the output of *stdump* or *nm* into *c++filt*. *c++filt* is installed in the directory */usr/lib/c++*. For example,

```
nm a.out | /usr/lib/c++/c++filt
```

libmangle.a The library */usr/lib/c++/libmangle.a* provides a function **demangle(char *)** that you can invoke from your own program to output a readable form of a mangled name. This is useful if you want to write your own tool for processing the output of *nm*, for example. You need to include the declaration

char * demangle(char *);

in your program, and link with the library with the *-lmangle* option.

size The *size* tool prints information about the text, rdata, data, sdata, bss, and sbss sections of the specific object or archive file. The contents and format of section data are described in Chapter 10 of the *Assembly Language Programming Guide*.

elfdump The *elfdump* tool lists the contents (including the symbol table and header information) of an ELF-format object file. See the *elfdump(1)* reference page for more information.

stdump The *stdump* tool outputs a file of intermediate-code symbolic information to standard out for *-32* executables only (for *-n32* and *-64*, use *dwarfdump*). See the *stdump(1)* reference page for more information.

Chapter 3

C++ Dialect Support

This chapter describes the C++ language implemented by the 7.0 compiler.

Note: This chapter applies to the -n32/-64 compiler only; the -32 compiler accepts an older version of the C++ language.

This chapter contains the following major sections:

- "About the Front End" on page 22 contains background information on the Silicon Graphics C++ front end.
- "New Language Features" on page 22 contains a list of features that are not in the *Annotated C++ Reference Manual* (ARM), are listed in the X3J16/WG21 Working Paper, and are supported in the Silicon Graphics C++ compiler.
- "Non-implemented Language Features" on page 24 contains a list of features that are not in the ARM, are listed in the X3J16/WG21 Working Paper, and are not supported in the Silicon Graphics C++ compiler.
- "Anachronisms Accepted" on page 25 contains a list of the anachronisms that are supported in this compiler when the -*anarch* option is enabled.
- "Extensions Accepted in Default Mode" on page 26 contains a list of the extensions that are accepted by the compiler by default.
- "Extensions Accepted in Cfront Compatibility Mode" on page 27 contains a list of extensions that are accepted by the compiler in *cfront* compatibility mode.
- "Cfront Compatibility Restrictions" on page 32 contains a list of the constructs that *cfront* supports but the Silicon Graphics compilers reject.

About the Front End

The front end accepts the C++ language as defined by *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup, Addison-Wesley, 1990, including templates, exceptions, and the anachronisms discussed in this chapter.

The front end also has a *cfront* compatibility mode (enabled by the *-cfront* option), which duplicates a number of features and bugs of *cfront*. Complete compatibility is not guaranteed or intended—the mode is there to allow programmers who have unwittingly used *cfront* features to continue to compile their existing code. By default, the front end does not support *cfront* compatibility. See "Cfront Compatibility Restrictions" for details.

The command line option *-anach* enables anachronisms. By default, anachronisms are disabled. See "Anachronisms Accepted" for details.

By default, the front end accepts certain extensions to the C++ language; these extensions will be flagged as warnings if you use the *-ansiW* option, and as errors if you use the *-ansiE* option. See "Extensions Accepted in Default Mode" for details.

New Language Features

The following features not in the ARM but in the X3J16/WG21 Working Paper are accepted:

- The dependent statement of an **if**, **while**, **do**-**while**, or **for** is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an **if**, **while**, **do**-**while**, or **for**, as the first operand of a **?** operator, or as an operand of the **&&**, `||`, or **!|** operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- Use of a global-scope qualifier in member references of the form **x.::A::B** and **p->::A::B** is allowed.
- The precedence of the third operand of the **?** operator is changed.
- If control reaches the end of the *main()* routine, and *main()* has an integral return type, it is treated as if a `return 0;` statement were executed.

- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form **A()** can be used even if **A** is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have non-type template parameters.
- A reference to **const volatile** cannot be bound to an rvalue.
- Qualification conversions, such as conversion from **T**** to **T const** * **const** * are allowed.
- Digraphs are recognized.
- Operator keywords (for example, **and**, **bitand**, and so on) are recognized.
- Static data member declarations can be used to declare member constants.
- **wchar_t** is recognized as a keyword and a distinct type.
- **bool** is recognized.
- Runtime type identification (RTTI), including **dynamic_cast** and the **typeid** operator, is implemented.
- Declarations in tested conditions (in **if**, **switch**, **for**, and **while** statements) are supported.
- Array **new** and **delete** are implemented.
- New-style casts (**static_cast**, **reinterpret_cast**, and **const_cast**) are implemented.

Non-implemented Language Features

The following features not in the ARM but in the X3J16/WG21 Working Paper are not accepted:

- Virtual functions in derived classes may not return a type that is the derived-class version of the type returned by the overridden function in the base class.
- **enum** types are not considered to be non-integral types.
- It is not possible to overload operators using functions that take **enum** types and no class types.
- Definition of nested classes outside of the enclosing class is not allowed.
- The new lookup rules for member references of the form **x.A::B** and **p**->**A::B** are not yet implemented.
- Classes are not assumed to always have constructors, and the distinction between trivial and nontrivial constructors is not implemented.
- **mutable** is not implemented.
- Namespaces are not implemented.
- **enum** types cannot contain values larger than can be contained in an **int**.
- Type qualifiers are not retained on rvalues (in particular, on function return values).
- **reinterpret_cast** does not allow casting a pointer to member of one class to a pointer to member of another class if the classes are unrelated.
- Explicit qualification of template functions is not implemented.
- Explicit instantiation of templates in the style of N0274/93-0067 is not implemented.
- Member templates are not implemented.
- Name binding in templates in the style of N0288/93-0081 is not implemented.
- The scope of a variable declared in a **for** loop is still the whole surrounding scope, not just the loop.
- In a reference of the form `f()->g()`, with **g** a static member function, **f()** is not evaluated. This is as required by the ARM, but the Working Paper, requires that **f()** be evaluated.
- **(p**->***pm)** = **0** cannot yet be written as **p**->***pm** = **0** (the syntax still matches the ARM and *cfront*).

- **typename** in templates is not implemented.
- Non-converting constructors are not implemented.
- Class name injection is not implemented.
- Overloading of function templates (partial specialization) is not implemented.
- Partial specialization of class templates is not implemented.
- Placement delete is not implemented.
- Putting a **try catch** around the initializers and body of a constructor is not implemented.
- The notation **:: template** (and ->**template**, and so forth) is not implemented.

Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled (via the *-anach* option):

- **overload** is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array **delete** operation. The value is ignored.
- A single **operator++()** and **operator--()** function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- A reference to a non-**const** type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-**const** class type may be initialized from an rvalue of the class type or a derived class thereof. No additional temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is

not applied to parameter types of such functions when the check for compatibility is performed, so that the following declares the overloading of two functions named **f**:

```
int f(int);
int f(x) char x; { return x; }
```

Note: In C this code is legal but has a different meaning: a tentative declaration of **f** is followed by its definition.

- A reference to a non-**const** class can be bound to a class rvalue of the same type or a derived type thereof.

```
struct A {
  A(int);
  A operator=(A&);
  A operator+(const A&);
};
main () {
  A b(1);
  b = A(1) + A(2); // Allowed as anachronism
}
```

Extensions Accepted in Default Mode

The following extensions are accepted by default (they can be flagged as errors or warnings by using *-ansiE* or *-ansiW*):

- A **friend** declaration for a class may omit the **class** keyword, as in the following:

```
class A {
  friend B; // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes, as in the following:

```
class A {
  const int size = 10;
  int a[size];
};
```

- In the declaration of a class member, a qualified name may be used, as in the following:

```
struct A {
  int A::f(); // Should be int f();
};
```

- **operator()** functions may have default argument expressions. A warning is issued.
- The preprocessing symbol **c_plusplus** is defined in addition to the standard **__cplusplus**.
- A pointer to a constant type can be **delete**d.
- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a default assignment operator—that is, such a declaration blocks the implicit generation of a copy assignment operator. (This is *cfront* behavior that is known to be relied upon in at least one widely-used library.) Here's an example:

  ```
  struct A { };
  struct B : public A {
    B& operator=(A&);
  };
  ```

  By default, as well as in *cfront*-compatibility mode, there will be no implicit declaration of **B::operator=(const B&)**, whereas in strict-ANSI mode **B::operator=(A&)** is not a copy assignment operator and **B::operator=(const B&)** is implicitly declared.

Extensions Accepted in Cfront Compatibility Mode

The following extensions are accepted in *cfront* compatibility mode (via the -*cfront* option):

- Type qualifiers on the **this** parameter may to be dropped in contexts such as the following example:

  ```
  struct A {
    void f() const;
  };
  void (A::*fp)() = &A::f;
  ```

  This is actually a safe operation. A pointer to a **const** function may be put into a pointer to non-**const**, because a call using the pointer is permitted to modify the object and the function pointed to will actually not modify the object. The opposite assignment would not be safe.
- Conversion operators specifying conversion to **void** are allowed.

- A non-standard friend declaration may introduce a new type. A friend declaration that omits the elaborated type specifier is allowed in default mode, but in *cfront* mode the declaration is also allowed to introduce a new type name.

```
struct A {
  friend B;
};
```

- The third operator of the **?** operator is a conditional expression instead of an assignment expression as it is in the current X3J16/WG21 Working Paper.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example,

```
int *p;
const int *&r = p;  // No temporary used
```

- A reference may be initialized with a null.
- Because *cfront* does not check the accessibility of types, access errors for types are issued as warnings instead of errors.
- When matching arguments of an overloaded function, a **const** variable with value zero is not considered to be a null pointer constant.
- No warning is issued when an **operator()** function has default argument expressions.
- An alternate form of declaring pointer-to-member-function variables is supported. Consider the following code sample:

```
struct A {
  void f(int);
  static void f(int);
  typedef void A::T3(int);  // nonstd typedef decl
  typedef void T2(int);     // std typedef
};
typedef void A::T(int);  // nonstd typedef decl
T* pmf = &A::f;          // nonstd ptr-to-member decl
A::T2* pf = A::sf;       // std ptr to static mem decl
A::T3* pmf2 = &A::f;     // nonstd ptr-to-member decl
```

where **T** is construed to name a routine type for a non-static member function of class **A** that takes an **int** argument and returns **void**; the use of such types is restricted to nonstandard pointer-to-member declarations. The declarations of **T** and **pmf** in combination are equivalent to a single standard pointer-to-member declaration, such as in the following example:

```
void (A::* pmf)(int) = &A::f;
```

A nonstandard pointer-to-member declaration that appears outside a class declaration, such as the declaration of **T**, is normally invalid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as **A::T3**, this feature changes the meaning of a valid declaration. *cfront* version 2.1 accepts declarations, such as **T**, even when **A** is an incomplete type; so this case is also excepted.

- Protected member access checking is not done when the address of a protected member is taken. For example:

```
class B { protected: int i; };
class D : public B { void mf(); };
void D::mf() {
  int B::* pmi1 = &B::i;  // error, OK in cfront mode
  int D::* pmi2 = &D::i;  // OK
}
```

Note: Protected member access checking for other operations (in other words, everything except taking a pointer-to-member address) is done in the normal manner.

- The destructor of a derived class may implicitly call the private destructor of a base class. In default mode this is an error but in *cfront* mode it is reduced to a warning. For example:

```
class A {
      ~A();
};
class B : public A {
      ~B();
};
B::~B(){}    // Error except in cfront mode
```

- When disambiguation requires deciding whether something is a parameter declaration or an argument expression, the pattern *type-name-or-keyword (identifier...)* is treated as an argument. For example:

```
class A { A(); };
double d;
```

```
A x(int(d));
A(x2);
```

By default **int(d)** is interpreted as a parameter declaration (with redundant parentheses), and **x** is a function; but in *cfront*-compatibility mode **int(d)** is an argument and **x** is a variable.

The declaration **A(x2)**; is also misinterpreted by *cfront*. It should be interpreted as the declaration of an object named **x2**, but in *cfront* mode is interpreted as a function style cast of **x2** to the type **A**.

Similarly, the declaration

```
int xyz(int());
```

declares a function named **xyz**, that takes a parameter of type "function taking no arguments and returning an **int**." In *cfront* mode this is interpreted as a declaration of an object that is initialized with the value **int()** (which evaluates to zero).

- A named bit-field may have a size of zero. The declaration is treated as though no name had been declared.
- Plain bit fields (in other words, bit fields declared with type **int**) are always unsigned.
- The name given in an elaborated type specifier is permitted to be a **typedef** name that is the synonym for a class name, for example:

```
typedef class A T;
class T *pa;               // No error in cfront mode
```

- No warning is issued on duplicate size and sign specifiers.

```
short short int i;  // No warning in cfront mode
```

- Virtual function table pointer update code is not generated in destructors for base classes of classes without virtual functions, even if the base class virtual functions might be overridden in a further-derived class. For example:

```
struct A {
  virtual void f() {}
  A() {}
  ~A() {}
};
struct B : public A {
  B() {}
  ~B() {f();}  // Should call A::f according to ARM 12.7
};
struct C : public B {
```

```
  void f() {}
} c;
```

In *cfront* compatibility mode, **B::~B** calls **C::f**.

- An extra comma is allowed after the last argument in an argument list, as for example in

```
f(1, 2, );
```

- A constant pointer-to-member-function may be cast to a pointer-to-function. A warning is issued.

```
struct A {int f();};
main () {
  int (*p)();
  p = (int (*)())A::f;  // Okay, with warning
}
```

- Arguments of class types that allow bitwise copy construction but also have destructors are passed by value (in other words, like C structures), and the destructor is not called on the “copy.” In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns.

  Note: Because the argument is passed differently (by value instead of by address), code like this compiled in *cfront* mode is not calling-sequence compatible with the same code compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.

- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.

- When an unnamed class appears in a **typedef** declaration, the **typedef** name may appear as the class name in an elaborated type specifier. For example:

```
typedef struct { int i, j; } S;
struct S x; // No error in cfront mode
```

- A typedef name may be used in an explicit destructor call. For example:

```
struct A { ~A(); };
typedef A B;
int main() {
  A *a;
  a->~B();  // Permitted in cfront mode
}
```

Cfront Compatibility Restrictions

Even when you specify the *-cfront* option, the Silicon Graphics C++ compilers are not completely backwards-compatible with *cfront*. The source constructs that *cfront* ignores but the Silicon Graphics compilers reject are listed below:

- Assignment to **this** in constructors and destructors is not allowed.
- If a C++-style (`//`) comment line is terminated with a backslash, the Silicon Graphics compiler will (correctly) continue the comment line into the next source line. (*cfront*, which uses the standard UNIX *cpp*, incorrectly terminates the comment at the end of the line.)
- You must have an explicit declaration of a constructor or destructor in the class if there is an explicit definition of it outside the class.
- You may not delete a pointer to a const.
- You may not pass a pointer to volatile data to a function that is expecting a pointer to non-volatile data.
- The Silicon Graphics compiler does not disambiguate between overloaded functions with a **char*** and **long** parameter, respectively, when called with an expression that is a 0 cast to a **char** type.
- You may not use redundant type specifiers.
- When in a conditional expression, the Silicon Graphics compiler does not convert a pointer to a class to an accessible base class of that class.
- You may not assign a comma-expression ending in a literal constant expression “`0`” to a pointer; the “`0`” will be treated as an **int**.
- You must not use the same identifier for more than one formal argument in a function definition.
- The Silicon Graphics compiler will *mangle* member functions declared as extern “C” differently from *cfront*. *CC* does not strip the type signature when you are building the mangled name. If you try to do so, you will see the following warning:

  ```
  Mangling of classes within an extern "C" block does not match
  cfront name mangling.
  ```

  You may not be able to link code containing a call to such a function with code containing the definition of the function that was compiled with *cfront*.

Chapter 4

Common Pitfalls

This chapter contains the following major sections:

- "Problems Involving C Linkage" discusses some problems you may encounter when you link your C++ programs to the C libraries.
- "Problems With Order of Specification of Libraries" on page 34 discusses some problems you may encounter when you order the libraries on the command line.

Problems Involving C Linkage

One of the most common problems you may encounter occurs when you link your C++ programs to C code (such as C libraries). This section contains many of the most typical problems you run into in that situation.

- Unexpected undefined symbols. You may see the following error message from the link-editor:

  ```
  Unresolved:  foo(int, char*)
  ```

  The presence of the prototype in this message indicates that this is a C++ function. Frequently this means not that the function **foo** is undefined, but that it is defined in a C object file or library, and the C++ declaration is missing an extern "C" linkage specification.

- Inconsistent linkage. You may see the following error message from the C++ front end (*fecc*):

  ```
  "afile.c", line 37: error (1311): linkage specification is
  incomplete with previous foo (declared at line 17)
  ```

  This means that two declarations for **foo()** were found with the same prototype, the first outside an extern "C" specification and the second inside. For example, you may have the following code, all in one compilation unit:

  ```
  void foo(int, char*);
  ......
  extern "C" {  void foo(int, char*);  }
  ```

Frequently these two declarations come from different header files.

- A “Too much C linkage” error. For example, you may see the following error message from the C++ front end (*edgcpfe*):

  ```
  "afile.C", line 37: error (3419): more than one instance of
  overloaded function "foo" has "C" linkage.
  ```

  This indicates that two declarations for **foo()** were found within extern “C” specifications but with different prototypes. Typically this happens when a function is declared in two header files with the wrong prototype in one of them, or when a function already declared in an included header file is redeclared incorrectly.

Problems With Order of Specification of Libraries

This section covers two typical problems you may encounter when you specify the order of your libraries.

- Inability to use the Silicon Graphics fast malloc routines, **malloc(3x)**.

  A related problem occurs with a command such as the following:

  ```
  CC foo.c++ -lmalloc
  ```

  The command mysteriously fails to use the “fast” *libmalloc.a* versions of **malloc()** and **free()**. Here again the order of libraries is

  ```
  -lmalloc -lC -lc
  ```

  At the time *ld* processes *libmalloc.a*, there are no undefined references to **malloc()** and **free()** (unless explicitly referenced from *foo.c++*). Only when **new** and **delete** are picked up from *libC.a* are **malloc()** and **free()** required, and then it is too late: their references have already been resolved from *libc.a* instead of *libmalloc.a*. Again, once the problem is recognized, the solution is easy. Just change the command to the following:

  ```
  CC foo.c++ -lC -lmalloc
  ```

- Mixing *stdio* and *iostreams*.

  If you mix *iostream* output using **cout** with **stdio** output using **printf**, and you are not careful about flushing the output buffers, you may see unexpected results. For example, consider the following program:

  ```
  #include <stdio.h>
  #include <ostream.h>
  main() {
  ```

```
  cout << "cout1\n";
  printf("printf1\n");
  cout << "cout2\n";
  printf("printf2\n");
 }
```

This code produces the following output:

```
printf1
printf2
cout1
cout2
```

This is because **cout** and **printf** use distinct buffers, and insertion of a newline into **cout** does not flush the buffer. To flush the buffer, you can insert the manipulator **flush** into the stream in the following way:

```
cout << "cout1\n" << flush;
```

You can also use the manipulator **endl** to insert a newline and flush as follows:

```
cout << "cout1" << endl;
```

On the other hand, consider the following program

```
main() {
  cout << "cout1 " << flush;
  printf("printf1 ");
  cout << "cout2 " << flush;
 }
```

This code produces the output

```
cout1 cout2 printf1
```

This is because the buffer for **printf** is not flushed until the program terminates. Here you need to call **fflush** after the call to **printf** as follows:

```
fflush(stdout);
```

This generates the following “expected” output:

```
cout1 printf1 cout2
```

If you wish to avoid explicitly flushing the buffer, you may insert the following code before performing any input/output:

```
ios::sync_with_stdio();
```

Chapter 5

Using Templates

This chapter discusses the Silicon Graphics C++ implementation of templates. It compares the Silicon Graphics implementation to those of the Borland® C++ and *cfront* compilers. It contains the following major sections:

- "Template Instantiation" describes how to perform template instantiation in the Silicon Graphics C++ environment.
- "How to Transition From cfront" on page 50 describes how a programmer currently using the cfront template instantiation mechanism can transition to the template instantiation scheme used by the new Silicon Graphics C++ compilers.
- "Template Language Support" on page 53 describes the language features for templates supported in the Silicon Graphics C++ environment, but not in *cfront*.

Template Instantiation

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately. The reasons for this are given below.

- You should have only one copy of each instantiated entity across all the object files that make up a program. (This applies to entities with external linkage.)
- You may write a specialization of a template entity. (For example, you can write a version of **Stack<int>**, or of just **Stack<int>::push**, that replaces the template-generated version. Often, this kind of specialization is a more efficient representation for a particular data type.) When compiling a reference to a template entity, the compiler does not know if a specialization for that entity will be provided in another compilation. The compiler cannot do the instantiation automatically in any source file that references it.

- You may not compile template functions that are not referenced. Such functions might contain semantic errors that would prevent them from being compiled. A reference to a template class should not automatically instantiate all the member functions of that class.

Note: Certain template entities are always instantiated when used (for example, inline functions).

If the compiler is responsible for doing all the instantiations automatically, it can do so only on a program-wide basis. The compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

By default, *CC* performs automatic instantiation at link time. It is also possible for you to instantiate all necessary template entities at compile time using the *-ptused* option. See "Explicit Instantiation" on page 41 for further details.

Automatic Instantiation

Automatic instantiation enables you to compile source files to object code, link them, run the resulting program, and never worry about how the necessary instantiations are done.

CC requires that for each instantiation you have a normal, top-level, explicitly-compiled source file that contains both the definition of the template entity and any types required for the particular instantiation.

Meeting Instantiation Requirements

You can meet the instantiation requirements in several ways:

- You can have each header file that declares a template entity contain either the definition of the entity or another file that contains the definition.
- When the compiler sees a template declaration in a header file and discovers a need to instantiate that entity, you can give it permission to search for an associated definition file having the same base name and a different suffix. The compiler implicitly includes that file at the end of the compilation. This method allows most programs written using the *cfront* convention to be compiled. See "Implicit Inclusion" on page 41.

- You can make sure that the files that define template entities also have the definitions of all the available types, and add code or **pragmas** in those files to request instantiation of the entities there.

Automatic Instantiation Method

1. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation.
2. When the object files are linked, a program called the prelinker is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.
3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it assigns the instantiation to it. The set of instantiations assigned to a given file, say *abc.C*, is recorded in an associated *.ii* file (for example, *abc.ii*). All *.ii* files are stored in a directory named *ii_files* created within your object file directory.
4. The prelinker then executes the compiler again to recompile each file for which the *.ii* file was changed. (The *.ii* file contains enough information to allow the prelinker to determine which options should be used to compile the same file.)
5. When the compiler compiles a file, it reads the *.ii* file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).
6. The prelinker repeats steps 3-5 until there are no more instantiations to be adjusted.
7. The object files are linked.

Details of Automatic Instantiation

Once the program has been linked correctly, the *.ii* files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the *.ii* files and do the indicated instantiations as it does the normal compilations. Except in cases where the set of required instantiations changes, the prelink step will find that all the necessary instantiations are present in the object files and that no instantiation assignment adjustments need be done. This is true even if the entire program is recompiled.

If you provide a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If the programmer adds a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper *.ii* file.

The *.ii* files should not, in general, require any manual intervention. The only exception is if the conditions below are all met.

- A definition is changed in such a way that some instantiation no longer compiles (it generates errors).
- A specialization is simultaneously added in another file
- The first file is recompiled before the specialization file and is generating errors.

The *.ii* file for the file generating the errors must be deleted manually to allow the prelinker to regenerate it.

If the prelinker changes an instantiation assignment, it will issue a message:

```
C++ prelinker: f__10A__pt__2_iFv assigned to file test.o
C++ prelinker: executing: usr/lib/DCC/edg-prelink -c test.c
```

The name in the message is the mangled name of the entity. These messages are printed if you use the *-ptv* option.

The automatic instantiation scheme can coexist with partial explicit control of instantiation by the programmer, through the use of pragmas or command-line specification of the instantiation mode.

The automatic instantiation mode can be disabled by using the *-no_prelink* option.

If automatic instantiation is turned off,

- the extra information about template entities that could be instantiated in a file is not put into the object file
- the *.ii* file is not updated with the command line
- the prelinker is not invoked

Implicit Inclusion

For the best results, you must include all the template implementation files in your source files. Since most *cfront* users do not do this, the compiler attempts to find unincluded template bodies automatically. For example, suppose that the following conditions are all true.

- template entity *ABC::f* is declared in file *xyz.h*
- an instantiation of *ABC::f* is required in a compilation
- no definition of *ABC::f* appears in the source code processed by the compilation

In this case, the compiler looks to see if the source file *xyz.n* exists. (By default, the list of suffixes tried for *n* is *.c*, *.C*, *.cpp*, *.CPP*, *.cxx*, *.CXX*, and *.cc*.) If so, the compiler processes it as if it were included at the end of the main source file.

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done. Implicit inclusion can be disabled with the *-no_auto_include* option.

Explicit Instantiation

CC instantiates all templates at compile time if you use the *-ptused* option. The compiler produces larger object files because it stores duplicate instantiations in the object files. Since duplicate copies may not be removed by the linker, and may exist in the final executables, the use of the *-ptused* is being deprecated.

The *CC* template instantiation mechanism also correctly handles static data members when you use the *-ptused* option. Static data members that need to be dynamically initialized may be instantiated in multiple compilation units. However, the dynamic initialization takes place only once. This is implemented by using a flag which is set the first time a static data member is initialized. This flag prevents further attempts to initialize it dynamically.

The *-ptused* option is acceptable for most small- or medium-sized applications. There are some drawbacks listed below:

- Instantiating everything produces large object files.

- Although duplicate code is removed, the associated debug information is not removed, producing large executables.
- If you change a template body, you must recompile every file that contains an instantiation of this body. (The easiest way to do this is for you to use *make* in conjunction with the *-MDupdate* option. See the *CC(1)* reference page and "Limitations" on page 48 for more information.)
- If you plan on specializing a template function instantiation, you may have to set `#pragma do_not_instantiate` if it is likely that the compiler-generated instantiation will contain syntax errors.
- Data is not removed, so there are multiple copies of static data members.

You can exercise finer control over exactly what is instantiated in each object file by using pragmas and command-line options.

Command Line Options for Template Instantiation

You may use command-line options to control the instantiation behavior of the compiler. These options are divided into sets of related options, as shown below. You use one option from each category; options from the same category are not used together. (For example, you do not use *-ptnone* in conjunction with *-ptused*.)

- *-ptnone* (the default), *-ptused*, and *-ptall*
- *-prelink* (the default) and *-no_prelink*
- *-auto_include*, *-no_auto_include*
- *-ptv*

The command line options are listed below.

-ptnone
: None of the template entities are instantiated. If automatic instantiation is on (in other words, *-prelink*), any template entities that the prelinker instructs the compiler to instantiate are instantiated.

-ptused
: Any template entities used in this compilation unit are instantiated. This includes all static members that have template definitions. If you specify *-ptused*, automatic instantiation is turned off by default. If you enable automatic instantiation explicitly (with *-prelink*), any additional template entities that the prelinker instructs the compiler to instantiate are also instantiated. The use of this option is being deprecated in the new compilers.

-ptall Any template entities declared or referenced in the current compilation unit are instantiated. For each fully instantiated template class, all its member functions and static data members are instantiated whether or not they are used. The use of this option is being deprecated in the new compilers.

Nonmember template functions are instantiated even if the only reference was a declaration. If you use *-ptall*, automatic instantiation is turned off by default. If you enable automatic instantiation explicitly (with *-prelink*), any additional template entities that the prelinker instructs the compiler to instantiate are also instantiated.

-prelink Instructs the compiler to output information from the object file and an associated *.ii* file to help the prelinker determine which files should be responsible for instantiating the various template entities referenced in a set of object files.

When *-prelink* is on, the compiler reads an associated *.ii* file to determine if any template entities should be instantiated. When *-prelink* is on and a link is being performed, the driver calls a “template prelinker.” If the prelinker detects missing template entities, they are assigned to files (by updating the associated *.ii* file), and the prelinker recompiles the necessary source files.

-no_prelink Instructs the compiler to not read a *.ii* file to determine which template entities should be instantiated. The compiler will not store any information in the object file about which template entities could be instantiated. This option also directs the driver not to invoke the template prelinker at link time.

This is the default mode if *-ptused* or *-ptall* are specified.

-auto_include Instructs the compiler to implicitly include template definition files if such definitions are needed. (See “Implicit Inclusion” on page 41.)

-no_auto_include
Disables implicit inclusion of template implementation files. (See “Implicit Inclusion” on page 41.)

-ptv Puts the template prelinker in verbose mode; when a template entity is assigned to a particular source file, the name of the template entity and source file is printed.

Note: In the case where a single file is compiled and linked, the compiler uses the -ptused option to suppress automatic instantiation.

Command Line Instantiation Examples

This section provides you with combinations of command line instantiation that you may want to use, along with an explanation of what these combinations would do, and what you might use them for.

Although there are many possible combinations of options, the most common are listed below:

`-ptnone -prelink -auto_include`
: This is the default mode, which is suitable for most applications. On the first build of an application, the prelinker determines which source files should instantiate the necessary template entities. On subsequent rebuilds, the compiler automatically instantiates the template entities.

`-ptused`
: This mode is suitable for small- and medium-sized applications. No prelinker pass is necessary. All referenced template entities are instantiated at compile time; the use of this option is being deprecated. Dynamically initialized static data members are also handled correctly (by using a runtime guard to prevent duplicate initialization of such members).

`-ptused -prelink`
: Use this combination when you have an archive or dynamic shared object (DSO) that has not been prelinked.

  When a DSO is built, it is automatically prelinked. When an archive is built, we recommend that you run the prelinker on the object files before archiving them. However, there are cases where a programmer may choose not to do so.

  For example, if an application is linked using multiple internal DSOs or archives, then you may choose not to prelink each DSO or archive, since that may create multiple instances of some template entities. When building an application using such archives or DSOs, you should use *-prelink* at compile time, even if the application is being built using *-ptused*. This is because the object files must contain not only instances of templates instances referenced in the compilation units, but also instances of template entities referenced in archives and DSOs.

-ptall -no_prelink
Use this combination when you are building a library of instantiated templates.

For example, consider if you have a "stack" template class containing various member functions. You may choose to provide instantiated versions of these functions for various common types (for example, int, float, and so on) and the easiest way of instantiating all member functions of a template is to use *-ptall*.

-ptnone -no_prelink
Use this combination if you are using template entities that are pre-instantiated.

For example, suppose you are using templates, but know that all of your referenced template entities have already been pre-instantiated in a library such as described in the previous example. In this case, you do not need any templates instantiated at compile time, and you should turn off automatic instantiation.

-auto_include Use this option if you are using template implementation files that are not explicitly included.

Most source code written for *cfront* style compilers does not usually include template implementation files, because the *cfront* prelinker does this automatically. The *-auto_include* option is the default mode, because you want to compile *cfront* style code, but still instantiate templates at compile time (which implies finding template implementation files automatically).

-no_auto_include
Use this option if you are using template implementation files that are explicitly included.

Source code written for compilers such as Borland/C++ includes all necessary template implementation files. Such source code should be compiled with the *-no_auto_include* option.

-ptnone -no_prelink
Use this combination if all your template instantiation is done through the use of pragmas.

By using these options, you guarantee that nothing will be instantiated unless an explicit pragma is provided.

Pragmas for Template Instantiation

You can use pragmas to control the instantiation of individual or sets of template entities. There are three instantiation pragmas:

instantiate Causes a specified entity to be instantiated.

do_not_instantiate
Suppresses the instantiation of a specified entity. Typically used to suppress the instantiation of an entity for which a specific definition is supplied.

can_instantiate
Allows (but does not force) a specified entity to be instantiated in the current compilation. You can use it in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity turns out to be required.

The arguments to the instantiation pragma may be

- a template class name, such as `A<int>`
- a member function name, such as `A<int>::f`
- a static data member name, such as `A<int>::i`
- a member function declaration, such as `void A<int>::f(int, char)`
- a template function declaration, such as `char* f(int, float)`

A pragma directive in which the argument is a template class name (for example, `A<int>`) is the same as repeating the pragma for each member function and static data member declared in the class.

When you instantiate an entire class, you may exclude a given member function or static data member using the **do_not_instantiate** pragma. See the example below:

```
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

You must present the template definition of a template entity in the compilation for an instantiation to occur. (You can also find the template entity with implicit inclusion.) If you request an instantiation by using the **instantiate** pragma and no template definition is available or a specific definition is provided, you will receive a link-time error. For example:

```
template <class T> void f1(T);
template <class T> void g1(T);
void f1(int) {}
void main()
{
          int i;
          double d;
          f1(i);
          f1(d);
          g1(i);
          g1(d);
}
#pragma instantiate void f1(int)
#pragma instantiate void g1(int)
```

f1(double) and **g1(double)** are not instantiated (because no bodies were supplied) but no errors are produced during the compilation. If no bodies are supplied at link time, you will receive a linker error.

You can use a member function name (for example, **A<int>::f**) as a pragma argument only if it refers to a single user-defined member function. (In other words, not an overloaded function.) Compiler-generated functions are not considered, so a name may refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists.

You can instantiate overloaded member functions by providing the complete member function declaration. See the example below:

```
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.

Specialization

CC supports *specialization*. In template instantiation, you specialize when you define a specific version of a function or static data member.

Because the compiler instantiates everything at compile time when the *-ptused* option is specified, a specialization is not seen until link time. The linker and runtime loader select the specialization over any non-specialized versions of the function or static data member.

See "Pragmas for Template Instantiation" on page 46 for information on how to suppress the instantiation of a function. You may find this useful if you intend to provide a specialization in another object file and the non-specialized version cannot be instantiated.

Building Shared Libraries and Archives

When you build a shared library or archive, you should usually instantiate any template instances that could be needed.

The prelinker is automatically run when building a shared library, but it must be run manually when building an archive. Follow the steps below to build your archive.

1. Enter the command `/usr/lib/DCC/edg_prelink a.o b.o`

   This instantiates any templates needed by these object files.

2. Enter the command `ar cr libtest.a a.o b.o` to build the archive.

Limitations

There are some limitations on template instantiation in the Silicon Graphics C++ environment:

- A template specialization that exists in an archive may fail to be selected.

  If you define a specialization within an object file that exists in an archive, and that object file does not satisfy any references (other than the reference to the specialization), then the object file is not selected. Any function generated from a template that appears before the archive will be used, although a specialization should take precedence over a generated function.

The following conditions have to be present for the bug to occur:

- A template member needs to be specialized.
- The specialization must live in an archive element.
- A non-specialization of the template member must live in an object file seen by the linker. For a non-specialization to live in an object file, *-ptused* must have been specified (in other words, not the default mode).
- Nothing else that exists in the archive element is referenced; that is, the specialization is probably the only thing in the object file.

You can use either of the following two workarounds:

- Force the archive element to be loaded by defining some dummy global within it, and passing the *-u* option to the linker to force an undefined reference to the dummy global.
- Use a **.so** (that is, a dynamic shared object) instead of an archive. The runtime loader will correctly select specializations from dynamic shared objects.

• There is no link time mechanism to detect changes in template implementation files or to re-instantiate those template bodies that are out of date when you use the *-ptused* option.

Since *Makefiles* usually makes object files dependent on the *.h* files where templates are defined, *make* may not enable you to rebuild the right set of object files if you modify a template implementation file. To make sure you rebuild all files that instantiate a given template when the template body changes, you must follow the steps below.

1. Use the *-MDupdate* option at compile time to update a dependency file (usually called *Makedepend*). The compiler lists dependencies for all applicable **#include** files, including template implementation files that are implicitly included.
2. Make sure that your *Makefile* includes this dependency file. See the *CC(1)* and *make* reference pages for more information on how to include files within a *Makefile*.

- The only object files that the prelinker can recompile are object files that have not been renamed after they were originally compiled. In particular, the following limitations apply:
  - The prelinker cannot recompile any object file that exists in an archive, since putting an object file in an archive is equivalent to renaming it. It is recommended that you run the prelinker on object files before putting them in an archive. A similar restriction applies to dynamic shared objects (see "Building Shared Libraries and Archives" on page 48).
  - The prelinker cannot compile an object file if it was renamed after being compiled. For example, consider the following command line:

    ```
    yacc gram.y CC -c y.tab.c mv y.tab.o object.o
    ```

    The prelinker does not know how to recompile *object.o*. If *object.o* contains unresolved template references that will not be satisfied by any other objects, you must use the *-ptused* option when compiling, or explicitly invoke the prelinker on the object file before moving it.

How to Transition From cfront

If you have compiled your source code with *cfront*, you may have to modify your build scripts to ensure that your templates are instantiated properly. This section discusses how to transition templates from *cfront* to the Silicon Graphics environment.

Mapping Template Options From cfront to CC

The *cfront* template-related options, their meaning, and the equivalent *CC* options are listed below:

-pta — Instantiates a whole template class rather than only those members that are needed. If you use automatic instantiation, there is no equivalent option for *CC*. If you use explicit instantiation, the *-ptall* option performs roughly the same action.

-pte suffix — Uses *suffix* as the standard source suffix instead of *.c*. There is currently no equivalent *CC* option. *CC* always looks for the following suffixes when looking for a template body to implicitly include: *.c, .C, .cpp, .CPP, .cxx, .CXX, .cc, .c++*.

-ptn Changes the default instantiation behavior for one-file programs to that of larger programs, where instantiation is broken out separately and the repository updated. One-file programs normally have instantiation optimized so that instantiation is done into the application object itself. There is currently no equivalent *CC* option.

One way of approximating this behavior is to compile your file with *-c*, and then link it, instead of compiling and linking in a single step. Another method is to create an empty dummy file, and compile/link your original file and the new dummy file in a single step. For example, you can use the following command line:

```
CC file.c dummy.c
```

-ptrpathname Specifies a repository, with *./ptrepository* as the default. If several repositories are given, only the first is writable, and the default repository is ignored unless explicitly named. There is no equivalent option for *CC*. The *cfront* "repositories" contain two kinds of information:

- information about where types and templates are defined
- object files containing template instantiations

The *CC* template instantiation mechanism does not use separate object files for template instantiations; all necessary template instantiations are performed in files that are part of the application (or library) being built. Information about which templates are capable of being instantiated by each file are embedded in the object file itself. This means that no repositories are needed. See "What to Do If You Use Object Files From cfront's Repository" and "What to Do If You Use Multiple Repositories" on page 52 for further information.

-pts Splits instantiations into separate object files, with one function per object (including overloaded functions), and all class static data and virtual functions grouped into a single object. There is no equivalent *CC* option. You can exercise fine-grained control over exactly which templates are instantiated in each file by using the instantiation pragmas described in "Pragmas for Template Instantiation" on page 46.

-ptv Turns on verbose or verify mode, which displays each phase of instantiation as it occurs, together with the elapsed time in seconds that phase took to complete. You should use this option if you are new to templates. Verbose mode displays the reason an instantiation is done and the exact *CC* command used. The *-ptv* option is also supported by

CC, and provides verbose information about the operation of the prelinker. The prelinker indicates which template instantiations are being assigned to which files, and which files are being recompiled.

What to Do If You Use Object Files From cfront's Repository

If you are used to the *cfront* template instantiation mechanism you may sometimes explicitly reference object files in the repository. This is often done when building an archive or a shared library. The general idea is to link a fake main program with a set of object files so as to populate the repository with the necessary template instantiations. The object files that were linked, along with the object files in the repository, are stored in an archive, or linked into a shared library.

cfront users do this to build an archive or library which has no unresolved template references. *CC* users who wish to build archives and shared libraries where all template references have been resolved can do the following:

- If you are building a shared library, the *CC* driver will automatically run the prelinker on the set of object files being linked into the shared libraries. No further action is necessary on the part of the programmer.
- If an archive is being built, the prelinker needs to be run explicitly on the object files, before invoking *ar*. See "Building Shared Libraries and Archives" on page 48 for information on how to do this.

What to Do If You Use Multiple Repositories

If you use the *cfront* template instantiation mechanism, you may sometimes use multiple repositories. For example, you may have an application which consists of multiple libraries. Each library is built in its own directory, and has its own repository. When you build the library, template functions are not instantiated. When the application is linked against these libraries, the necessary templates are instantiated at link time. The repositories provide enough information about where to find the necessary template declarations and implementations.

CC does not use repositories, and you can use various strategies when linking a set of object files against a set of libraries that contain references to uninstantiated template functions. Some examples are given below:

- If it is possible that all uninstantiated template functions can be instantiated in the object files being linked into the application, the prelinker will do so automatically. However, it is possible that a library uses a template internally, which is never used by the object files being linked into the application. Such templates are not instantiated by the prelinker, resulting in undefined symbols.
- A better strategy is to prelink each library when it is built, so that the main program is not burdened with having to perform these instantiations. One problem occurs if multiple libraries use the same template functions: if each library is prelinked, multiple copies of such functions will be generated. Removal of duplicate functions takes place only in *.o* and *.a* files; shared libraries cannot have any duplicate code removed.

Template Language Support

The language support for templates in the Silicon Graphics C++ environment is more extensive than for *cfront*. Some of the additional template language constructs supported by the Silicon Graphics C++ environment are listed below:

- You may use nested classes, typedefs, and enums in class templates, including variant typedefs and enums. (A variant member type depends on the template parameters in some way.)
- You may use floating point numbers, pointers to members, and more flexible specifications of constant addresses.
- You may use default arguments for class template non-type parameters. For example:

  ```
  template <int I = 1> class A {};
  ```

- You may allow a non-type template parameter to have another template parameter as its type. For example:

  ```
  template <class T, T t> class A {
  public:
    T              a;
    A(T init_val = t)  { a = init_val; }
  };
  ```

- You may use what are essentially template classes instantiated with the template parameters of other class or function templates.

```
template <class T, int I> struct A {
          static T b[I];
};

template <class T> void f(A<T,10> x) {}
template <class T> void f(A<T, 3> x) {}

void main()
{
          A<int,10> m;
          A<int,3> n;
          int i = f(m);
          int j = f(n);
}
```

  The function template would be considered tagged twice by *cfront*, and the code calls tagged ambiguous by the Borland/C++ compiler.

- You may use circular template references. For example:

```
template <class T> class B;
template <class T> class C;

template <class T> class A { B<T> *b; };
template <class T> class B { C<T> *c; };
template <class T> class C { A<T> *a; };

A<int> a;
```

  cfront generates an error on this code.

- *CC* is more consistent than other C++ compilers about where a class template name must be followed by template arguments. For example:

```
template <class T> struct X {
          X();
          ~X();
          X*x;
          int X::* x2;
          void f();
          void g(){ X x;}
};

struct X<char> {
```

```
          X();
          ~X();                // Borland error
          X*x;                 // Borland error
          int X::* x2;         // Borland error
          void f();
          void g(){ X x;}      // Borland error };

template <class T> void X<T>::f()
{
          X x;                 // cfront error }

void X<char>::f()
{
          X x;                 // cfront & Borland error
}

X<int> x;
X<char> xc;
```

cfront allows **X** to be used as a type name in the inline body of **g** but not in the out-of-line body of **f**. Borland/C++ uses one set of rules for class templates and a different set of rules for specializations. With *CC*, you may use **X** in all of the cases shown.

- You may use forward declarations of class specializations.
- You may use nested classes as type arguments of class templates.
- You may use default arguments for all types of function templates, including arguments based on template parameter types. For example:

```
template <class T> void f(T t, int i = 1) {}
template <class T> void f(T t, T   i = 1) {}
```

C and C++ Pragma Directives

This appendix discusses the behavior of each recognized #pragma directive available for C and C++ in the Silicon Graphics environment. The pragmas are the following:

#pragma weak *weak_symbol = strong_symbol*

The *weak_symbol* is an alias that denotes the same function or data object denoted by the *strong_symbol*, unless a defining declaration for the *weak_symbol* is encountered at static link time. If encountered, the defining declaration preempts the weak denotation.

You must define the *strong_symbol* within the same compilation unit in which the pragma occurs. You should also declare the *weak_symbol* with **extern** linkage in the same compilation unit. The **extern** declaration of the weak symbol is not required, unless the symbol is referenced within the compilation unit, but Silicon Graphics recommends it for type-checking purposes. The weak and strong symbols must be declared with compatible types. When the strong symbol is a data object, its declaration must be initialized.

Weak **extern** declarations are typically used to export non-ANSI C symbols from a library without polluting the ANSI C name-space. As an example, *libc* may export a weak symbol **read()**, which aliases a strong symbol **_read()**, where **_read()** is used in the implementation of the exported symbol **fread()**. You can either use the exported (weak) version of **read()**, or define your own version of **read()** thereby preempting the weak denotation of this symbol. This will not alter the definition of **fread()**, since it only depends on the (strong) symbol **_read()**, which is outside the ANSI C name-space.

Note: **#pragma weak** is not supported in *-32*.

#pragma once

This pragma has no effect in **–32** mode, but will ensure idempotent *include* files in **–64** mode (i.e. that an *include* file is included at most once in one compilation unit). Silicon Graphics recommends enclosing the contents of an include file *afile.h* with an **#ifdef** directive similar to:

```
#ifndef afile_INCLUDED
#define afile_INCLUDED
```

```
<contents of afile.h>
#endif
```

#pragma pack(*n*)

This pragma controls the layout of structure offsets, such that the strictest alignment for any structure member will be *n* bytes, where *n* is 0, 1, 2, 4, 8, or 16. When *n* is 0, the compiler returns to default alignment for any subsequent **struct** definitions.

A **struct** type defined in the scope of a **#pragma pack**(*n*) has at most an alignment of *n* bytes, and the packed characteristics of the type apply wherever the type is used, even outside the scope of the pragma in which the type was declared. The scope of a **#pragma pack** ends with the next **#pragma pack**, hence this pragma does not nest: There is no way to "return" from one instance of the pragma to a lexically earlier instance of the pragma.

A structure declaration must be subjected to identical instances of a **#pragma pack** in all files, or else misaligned memory accesses and erroneous struct member dereferencing may ensue.

Silicon Graphics strongly discourages the use of **#pragma pack**, since it is a nonportable feature and the semantics of this pragma may change in future compiler releases. Note that references to fields in **#packed structs** may be less efficient than references to fields in unpacked structs.

#pragma intrinsic(*a_function*)

This pragma allows certain preselected functions from *math.h*, *stdio.h*, and *string.h* to be inlined at a call-site for execution efficiency. The **#pragma intrinsic** has no effect on functions other than the preselected ones. Exactly which functions may be inlined, how they are inlined, and under what circumstances inlining occurs is implementation defined and may vary from one release of the compilers to the next. The inlining of intrinsics may violate some aspect of the ANSI C standard (e.g., the **errno** setting for *math.h* functions). All intrinsics are activated through pragmas in the respective standard header files and only when the preprocessor symbol __INLINE_INTRINSICS is defined and the appropriate include files are included. __INLINE_INTRINSICS is predefined by default only in **–cckr** and **–xansi** mode.

#pragma hdrstop

If *–pch* is on, **#pragma hdrstop** indicates the point at which the precompiled header mechanism snapshots the headers. If *–pch* is off, **#pragma hdrstop** is ignored. See the *Compiling and Performance Tuning Guide* for details on the precompiled header mechanism.

#pragma instantiate *declaration*

Causes a specified instance of a template declaration to be immediately instantiated at that spot. For example:

```
#pragma instantiate void List<int>::push(int)
```

The declaration needs to be a complete declaration of a function or a static data member, exactly as if you would have specified it for a specialization of the template.

#pragma do_not_instantiate *declaration*

The opposite of **#pragma instantiate**: if the compiler sees this pragma, it will not instantiate the specific declaration in this compilation unit, even if you use that instance in your code.

The MIPSpro compilers also silently recognize many commonly used pragmas; however, they have no effect. Some of these include:

#pragma no side effects(*a_function*)

Tells the compiler that a call to a function of the given name does not cause any modifications to objects accessible outside the function body. Such information can be useful for optimization and parallelization purposes.

#pragma ident *version*

Adds a *.comment* section in the object file and puts the revision string inside it.

#pragma *int_to_unsigned* **identifier**

Identifies *identifier* as a function whose type was **int** in a previous releases of the compilation system, but whose type is **unsigned int** in the MIPSpro compiler release. The declaration of the identifier must precede the pragma:

```
unsigned int strlen(const char*);
#pragma int_to_unsigned strlen
```

This declaration makes it possible for the compiler to identify where the changed type may affect the evaluation of expressions.

Glossary

Terms shown in *italics* indicate glossary items, as well as buttons, file names, and conventions.

access

The degree of privilege to a member of a class: *public*, *private*, or *protected*.

ARM

Acronym for *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup.

base class

Parent of a class.

data

In C++, data member of a class. A variable that contains state information for a class. A field of a class that is not a *method*.

database

Compiler-generated static analysis set of data built from a fileset in the static analyzer.

instantiate

In C++, to declare an object of type *classname*. The result is an instance or an object.

mangle

Encoding a function name to support overloading.

member function

C++ function that is a member of a class or structure data type. Also known as a method.

members

Either or both *data* and *method*s belonging to a class (class members).

method

C++ function that is a member of a class or structure data type. Also known as a member function.

private

In C++, access to the class member is restricted to the class in which it is defined, friend classes, or friend functions.

protected

In C++, access to the class member is restricted to the class (and all derived classes) in which it is defined, friend classes, and friend functions.

public

In C++, access is open to any method or function.

specialization

In template instantiation, defining a specific version of a function or static data member.

templates

A description of a class or function that is a model for a family of related classes or functions.

Index

Symbols

#pragma hdrstop, 59
#pragma ident version, 59
#pragma int_to_unsigned identifier, 59
#pragma intrinsic, 58
#pragma no_side_effect, 59
#pragma once, 57
#pragma pack, 58
#pragma weak, 57

Numbers

64-bit mode
  pragmas, 59

A

ABI, 64-bit changes, 2
ABI changes, 2
ABI changes, other, 8
accepted anachronisms, 25
anachronisms, accepted, 25
ANSI C, 57
assignment to this, 5

B

bool, 3
built-in bool type, 3
built-in w_char_t type, 4

C

C++ compilers
  6.0 versions, 2
  64 vs. 32 bit, 10
  using, 9
C++ environment, 1
c++patch, 16
catch, 6
CC -64
  command line, 11
CC command, options, 17
C compiler, 16
Cfront
  compatibility with Delta/C++, 12
cfront, compatibility restrictions, 32
cfront template transition, 50
changes, 64-bit ABI, 2
C linkage, problems with, 33
command lines
  CC -64, 11
  samples, 11
compatibility restrictions, cfront, 32

compiler, 16
compilers
  64 vs. 32 bit, 10
  using, 9
compiling, 13, 14
compiling, and linking, 13
complex arithmetic library, 12
contents of guide, xiii
conventions, font, for manual, xv

D

debugging, 12
delete, operator, 3
Delta/C++
  Cfront compatibility with, 12
dialect support, C++, 21
documentation, recommended reading, xiii

E

exception handling, 5
  catch, 6
  throw, 6
  try, 6
exctensions
  accepted default, 26
export
  non-ANSI, 57
extensions
  accepted cfront, 27
  cfront accepted, 27
  default accepted, 26
extern declarations, 57

F

fast *malloc*, 34
features
  new language, 22
  non-implemented, 24
features, new, 2
font conventions, for manual, xv
front end
  about, 22

G

global constructors, 16
guide contents, xiii

H

handling, exception, 5

I

implicit inclusion, 41
inclusion, implicit, 41
INLINE_INTRINSICS, 58
instantiation
  automatic, details of, 39
  automatic method of, 39
  requirements, 38
instantiation, command-line options, 42
instantiation, template, 41
iostream library, 12
iostreams, 34

L

language, new features, 22
ld, 16
libraries, 12, 17
libraries, problems with order of specification, 34
linkage
  problems with, 33
linkage, problems with C, 33
link editor, 16
linking, 13, 17
linking, and compiling, 13
link libraries, 17
loader, 16

M

malloc, 34
multi-language programs, 17

N

NCC
  mapping template options from cfront, 50
new, operator, 3

O

object files, 13
  linking, 17
  tools, 19
operators new and delete, 3
options, translator, 17

P

pragmas, 57
  for template instantiation, 46
  ignored, 59
ptrepository, object files in, 52

R

related information, xiv
repositories, multiple template, 52
RTTI, 7
runtime type identification, 7

S

shared libraries, building, 48
source file, suffix, 16
specialization, 48
stdio, 34
symbols unexpected undefined, 33

T

template instantiation
  archives, 48
  shared libraries, 48
templates
  automatic instantiation, 38
  command-line instantiation, 42
  instantiation, 41, 46
  instantiation examples, 44
  introduction to instantiation, 37
  language support, 53
  mapping cfront to NCC options, 50
  multiple repositories, 52

object files in cfronts ptrepository, 52
restrictions, 48
specialization, 48
transitioning from cfront, 50
this, assignment to, 5
throw, 6
tools, object files, 19
translator options, 17
troubleshooting, 33
try, 6
type identification, runtime, 7

U

unexpected undefined symbols, 33

W

wchar_t, 4
weak_signal=strong_symbol, 57

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-0704-110.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California 94043-1389