C++ Programmer's Guide

Document Number 007-0704-120

CONTRIBUTORS

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

C++ Programmer's Guide
Document Number 007-0704-120

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 xiv
Information Related to Silicon Graphics C++ xiv
C++ Language References xiv
C++ Parallelization References xv
The Standard Template Library xv
MIPSpro Development Environment References xv
Obsolete References xvi
Conventions Used in This Guide xvi

1. Understanding the Silicon Graphics C++ Environment 1
Silicon Graphics C++ Environment 2
Understanding ABIs and ISAs 3
N32, N64, and O32 Compilation 4
New Features of the MIPSpro 7.2 C++ Compilers 6
Namespaces 6
for Statement Scoping Rules 7
Member Templates 8
Partial Specialization of Class Templates 8
New Specialization Syntax 9
New Keywords 10

Other Features and ABI Changes in the MIPSpro C++ Compilers 10
Operators new[] and delete[] 11
Built-in bool Type 12
Built-in wchar_t Type 12
Assignment to this 13
Exception Handling 14
Run-time Type Identification 16
Other ABI Changes 17
Cfront Compatibility 17
C++ Libraries 18
Debugging 18

2. Compiling, Linking, and Running C++ Programs 19
Compiling and Linking 20
Translators and Drivers 20
Compilation 21
Sample Command Lines 23
Multi-Language Programs 24
Object File Tools 24

3. C++ Dialect Support 27
About the Front End 28
New Language Features 29
Non-implemented Language Features 31
Anachronisms Accepted 32
Extensions Accepted in Default Mode 33
Extensions Accepted in Cfront-Compatibility Mode 34
Cfront Compatibility Restrictions 38

4. Using Templates 41
Template Instantiation 42
Automatic Instantiation 43
Meeting Instantiation Requirements 43
Automatic Instantiation Method 43
Details of Automatic Instantiation 44
Implicit Inclusion 45
Explicit Instantiation 46
Command Line Options for Template Instantiation 47
Command Line Instantiation Examples 49
Pragmas for Template Instantiation 51
Specialization 53
Building Shared Libraries and Archives 53
Limitations on Template Instantiation 54
Unselected Template Specializations in Archives 54
Undetected Link-Time Changes in Template Implementation Files 55
Prelinker Cannot Recompile Renamed Files 55
How to Transition From Cfront 56
Mapping Template Options From Cfront to CC 56
What to Do If You Use Object Files From Cfront's Repository 58
What to Do If You Use Multiple Repositories 58
Template Language Support 59

A. C and C++ Pragma Directives 63
Common Recognized Pragmas 63
#pragma can_instantiate 64
#pragma do_not_instantiate 64
#pragma hdrstop 64
#pragma instantiate 64
#pragma int_to_unsigned 65
#pragma intrinsic 65
#pragma no side effects 65
#pragma once 66
#pragma pack 66
#pragma weak 67
Other References on Pragmas 68

B. MIPSpro 7.2 C++ Porting Errors 69
Error 1040 70
Error 1140 71
Error 1164 71
Error 1168 71
Error 1220 72
Error 1320 73
Error 1324 73
STL Error: Error 1278 74
Warning 1306 76

Glossary 79

Index 81

List of Examples

Example 1-1 Exception Handling 14
Example 1-2 Run-time Type Identification (RTTI) 16

List of Figures

Figure 1-1 Silicon Graphics C++ Environment 2
Figure 2-1 The MIPSpro and ucode C++ Compilation Processes 22

List of Tables

Table 1-1 Features of the Application Binary Interfaces 3
Table 1-2 ISAs and Targeted MIPS Processors 4
Table 1-3 ISAs, ABIs, and Their Host CPUs 5

About This Guide

This guide describes how to use the MIPSpro™ 7.2 release of the Silicon Graphics® C++ compiler environment. It discusses the native C++ compilers that produce 32-bit (O32), 64-bit (N64), and high-performance 32-bit (N32) objects.

What This Guide Contains

This guide contains the following chapters:

- Chapter 1, "Understanding the Silicon Graphics C++ Environment," describes the Silicon Graphics C++ environment.
- 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, "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.
- Appendix B, "MIPSpro 7.2 C++ Porting Errors" describes some of the errors that may arise when you port code to the 7.2 release.

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.

Information Related to Silicon Graphics C++

The following sections contain references to aid your understanding of the C++ language and the MIPSpro environment.

C++ Language References

At the time of the MIPSpro 7.2 release, there was no source available that completely described the MIPSpro C++ language. The following references are useful for understanding the C++ language.

Ellis, Margaret and Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Reading: Addison-Wesley Publishing Company, 1990. This is a seminal, but dated, reference.

Stroustrup, Bjarne. *The C++ Programming Language*, 3rd. ed. Reading: Addison-Wesley Publishing Company, 1997. A general reference for the C++ language, and very relevant to MIPSpro C++.

X3 Secretariat: *Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++*. X3J16/96-0018 WG21/N0836. American National Standards Institute (ANSI) Standards Secretariat: CBEMA, 1250 Eye Street NW, Suite 200, Washington, DC 20005. A recent version of the draft C++ standard.

C++ Parallelization References

For details about using manual and automatic parallelization with your C++ programs, refer to the following:

- The *C Language Reference Manual* contains, in Chapter 11, information about using the C and C++ manual multiprocessing pragmas to label parallel loops and code regions.
- *MIPSpro Automatic Parallelizer Programmer's Guide* discusses how to use the automatic parallelizer, introduced with the MIPSpro 7.2 compilers, to process Fortran, C, and C++ code for multiprocessor systems.

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 draft ANSI/ISO C++ standard. The library is shipped with the MIPSpro 7.2 C++ compilers. Documentation in html format is available under the SGI home page at http://www.sgi.com/Technology/STL.

MIPSpro Development Environment References

The following manuals provide information that is useful for understanding the Silicon Graphics C++ environment.

- *dbx User's Guide* discusses how to debug your code in the Silicon Graphics development environment with *dbx*.
- *Developer Magic: Debugger User's Guide* discusses the Debugger, a source-level debugging tool that is part of ProDev WorkShop. It is a suite of graphical, interactive, software engineering tools for C, C++, Fortran, and Ada.
- *MIPSpro Compiling and Performance Tuning Guide* discusses how to compile programs written in the Silicon Graphics development environment (C, Fortran, and C++), and tune their performance.
- *MIPSpro N32 ABI Handbook* describes the N32 high performance 32-bit application binary interface (ABI) for the MIPS® architecture.
- *MIPSpro 64-Bit Porting and Transition Guide* discusses the changes introduced with IRIX™ 6.2, and describes the MIPSpro 7.x O32, N64, and N32 compilers.

Obsolete References

As of the MIPSpro 7.2 release, Silicon Graphics no longer ships the following manuals because they describe an obsolete version of the C++ language.

- *C++ Language System Overview*
- *C++ Language System Product Reference Manual*
- *C++ Language System Library*

Conventions Used in This Guide

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

- **Bold**—Functions, option flags, keywords, and classes
- *Italics*—File names, field names, variables, emphasis, glossary terms, and IRIX commands
- Regular—Data types, and text
- `Fixed-width`—Code examples and command syntax

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 2 discusses the four C++ compilers Silicon Graphics provides for IRIX 6.x systems.
- "Understanding ABIs and ISAs" on page 3 describes the application binary interfaces and instruction set architectures supported by the MIPSpro compilers.
- "New Features of the MIPSpro 7.2 C++ Compilers" on page 6 describes the new features that are available in the 7.2 release of the MIPSpro C++ compilers.
- "Other Features and ABI Changes in the MIPSpro C++ Compilers" on page 10 describes the features and ABI changes that were available starting with the 6.2 and 7.0 versions of the MIPSpro C++ compilers.
- "Cfront Compatibility" on page 17 discusses the C++ code restrictions that are enforced by the Silicon Graphics C++ environment, but were not enforced by cfront.
- "C++ Libraries" on page 18 discusses the C++ libraries in the Silicon Graphics C++ environment.
- "Debugging" on page 18 discusses the Silicon Graphics C++ debugging environment.

Silicon Graphics C++ Environment

The Silicon Graphics 7.2 C++ environment provides four C++ compilers for IRIX 6.x systems. As shown in Figure 1-1, there are two MIPSpro C++ compilers, a 32-bit version and a 64-bit version. They are known as the N32 and N64 compilers because of the *application binary interfaces* (ABIs) they generate. These compilers accept a dialect of C++ that closely resembles the ANSI/ISO draft C++ standard (see "C++ Language References" on page xiv), and are strongly recommended by Silicon Graphics.

Note: In fact, the N32 and N64 compilers share the same front end (fecc) and code generator (be). These compilers can be viewed, alternatively, as one compiler that produces two different ABIs.

The 32-bit ucode C++ compiler (O32 ABI) is also available. The O32 compiler is an older compiler that is no longer being enhanced. It is included to support legacy code. Finally, the C++ compiler based on cfront is still available, although it is unsupported. It is provided to ease code migration. See the C++ release notes for the latest information about the status of these two compilers.

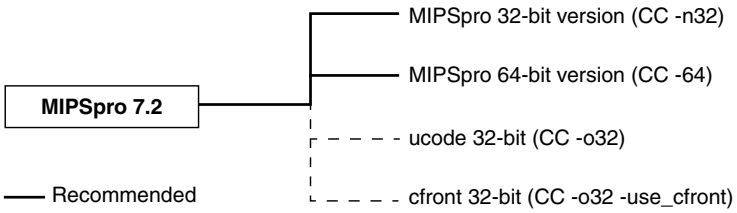

Figure 1-1 Silicon Graphics C++ Environment

The commands that invoke the four compilers are listed below:

CC -n32 32-bit native MIPSpro compiler. Generates N32 ABI objects.

CC -64 64-bit native MIPSpro compiler. Generates N64 ABI objects.

CC -o32 32-bit native ucode compiler. Generates O32 ABI objects.

CC -o32 -use_cfront
32-bit cfront compiler (*OCC*), based on C++ to C translation.

For more information about compilers and ABIs, see the cc(1), CC(1), and ABI(5) reference pages, or the *MIPSpro Compiling and Performance Tuning Guide*.

Understanding ABIs and ISAs

An application binary interface defines a system interface for executing compiled programs. Among the important features the ABI specifies are the following:

- supported *instruction set architectures* (ISAs)
- size of the address space
- object file formats
- calling conventions
- register size
- number of registers

The three ABIs that are relevant to MIPSpro 7.2 C++ are N32, N64, and O32. See Table 1-1 for a summary of the ABIs, their features, and their relationships to the MIPS ISAs.

Table 1-1 Features of the Application Binary Interfaces

	N32 (-n32)	**N64 (-64)**	**O32 (-o32)**
Default ISA	MIPS III	MIPS IV	MIPS II
Alternate ISA (Option)	MIPS IV (-mips4)	MIPS III (-mips3)	MIPS I (-mips1)
Floating Point Registers	32	32	16
Register Size	64 bits	64 bits	32 bits
int	32 bits	32 bits	32 bits
long int	32 bits	64 bits	32 bits
char*	32 bits	64 bits	32 bits

The instruction set architecture is the set of instructions recognized by a processor. It is the interface between the lowest level software and the processor. Table 1-2 shows the MIPS ISAs and the MIPS processors for which they were designed.

Table 1-2 ISAs and Targeted MIPS Processors

ISA	Target Processor
MIPS IV	R5000™, R8000®, R10000™
MIPS III	R4000® (Rev 2.2 and later), R4400®, R46000
MIPS II	R4000 (Rev. 2.1 and earlier)
MIPS I	R2000™, R3000®

Table 1-1 and Table 1-2 are intended to give an overview of ABIs and MIPS ISAs. They do not show details such as the default ISAs shown in Table 1-1 can be changed with the environmental variable *SGI_ABI*. Also, these tables do not indicate facts such as that an R10000-based, IRIX 6.3 O2™ system does not support N64 MIPS IV, while an R10000-based IRIX 6.4, OCTANE™ system does. The release notes for your system and the *MIPSpro Compiling and Performance Tuning Guide*, as well as the cc(1), CC(1), and ABI(5) references pages, give more information about these details.

N32, N64, and O32 Compilation

CC -64 and *CC -n32* both invoke the same MIPSpro front end, fecc. The fecc front end uses 64-bit pointers, addresses, and **long int**s for *CC -64*, and 32-bit pointers, addresses, and **long int**s for *CC -n32*. *CC -o32* has a different front end, edgcpfe, with 32-bit pointers, addresses, and **long int**s.

There are other differences between the two MIPSpro compilers and O32:

- The O32 compiler back-end (optimizer and code generator) is different from that of the N32 and N64 compilers.
- The code generated for N32 and N64 is much more highly optimized than that generated for O32. However, it may take longer to compile code for N32 and N64 due to the increased optimization performed.
- The warning options used by the **-woff** option are different.

The default compilation modes are the following

- *CC -64* is **-mips4**
- *CC -n32* is **-mips3**
- *CC -o32* is **-mips2**

Silicon Graphics recommends that most of your development be for the N32 ABI: **-n32 -mips3** for R4x00 systems and **-n32 -mips4** for R5000, R8000, and R10000 systems. This gives your program access to the full MIPS III instruction set for R4x00 systems. On systems that use the R5000 or above, it allows your program to use the MIPS IV instruction set with the lower overhead of a 32-bit address space. You can reserve the higher overhead **-64 -mips4** option for those applications that need the 64-bit address space on R8000 and R10000 systems.

The general relationship between ABIs, ISAs, and the CPUs that can run them is shown in Table 1-3. Again, because of system variations, there are some exceptions to the combinations shown. Consult your system's reference pages and release notes for further information.

Table 1-3 ISAs, ABIs, and Their Host CPUs

	-n32	**-64**	**-o32**
-mips4	R10000, R8000, R5000	R10000, R8000	
-mips3	R10000, R8000, R5000, R4600, R4400, R4000 (>=Rev. 2.2)		
-mips2			R10000, R8000, R5000, R4600, R4400, R4000
-mips1			R10000, R8000, R5000, R4600, R4400, R4000, R3000

Note: The objects of one ABI are incompatible with those of another; they cannot be linked together.

Other sources of information for O32, N32, and N64 compiling are the following:

- Refer to the *MIPSpro N32 ABI Handbook* for a primer on N32.
- Refer to the *MIPSpro Compiling and Performance Tuning Guide* for a more complete discussion on how to set up the IRIX environment for the MIPSpro compilers or O32.
- Refer to the *MIPSpro 64-Bit Porting and Transition Guide* for further information on N32 and N64 compilers.

New Features of the MIPSpro 7.2 C++ Compilers

This section describes the major new features of the MIPSpro 7.2 C++ compilers for N32 and N64. For potential problems that may arise when you port your code to the 7.2 compilers, see Appendix B, "MIPSpro 7.2 C++ Porting Errors."

Namespaces

MIPSpro 7.2 C++ implements the **namespace** mechanism, including **using** declarations and **using** directives. A *namespace* is a C++ device for denoting a logical grouping. It is a scope and follows the same scoping rules as global scopes, local scopes, and classes. A name in one namespace can be used in another namespace if the name is qualified by its original namespace. Declarations for namespaces **SpaceOne**, **SpaceTwo**, and **SpaceThree** might look like this:

```
namespace SpaceOne {
  double  func1();
  bool    func2( int ) { /*...*/}
}

namespace SpaceTwo {
  int    funcA();
  using  SpaceOne::func2; /* A using declaration */
}

namespace SpaceThree {
  double funcB();
  using  namespace SpaceOne; /* A using directive */
}
```

The declaration in the statement `using SpaceOne::func2;` makes **func2()** in **SpaceOne** available in **SpaceTwo**. The directive in `using namespace SpaceOne;` makes all of the names from **SpaceOne** available in **SpaceThree**.

In the MIPSpro 7.2 implementation, access declarations are broadened to match the corresponding **using** declarations. However, library names and routines that the draft C++ standard specifies to be in **std** namespace are actually in global namespace. For example, the standard exception **bad_cast,** which is specified to be in **std** namespace, has global scope. Such symbols cannot be referred to using the notation **std::**.

Note: The flag **-experimental**, which was used with the 7.1 compilers to invoke an version that supported the **namespace** mechanism, is not supported in MIPSpro 7.2.

for Statement Scoping Rules

The draft C++ standard specifies that the scope of a variable declared in the *for-init-statement* of a **for** loop is the scope of only the loop, not the scope of the block enclosing the loop. This feature is implemented in MIPSpro 7.2 C++. The following code is an example of the new scoping rule.

```
int f(int j) {
  int i = 0; {
    for (int i = 0; i < j; i++) { } // for-init-statement: int i = 0;
    return i;  // By new rule, 0 is returned.
  }
}
```

Because 7.2 is the first MIPSpro release to implement this rule, and because this change could break existing code, **-LANG:ansi-for-init-scope=on** must be specified on the command line to use this feature. Silicon Graphics strongly recommends that you compile with **-LANG:ansi-for-init-scope=on** to find and correct code that does not conform to this feature of the draft standard.

Member Templates

The member templates feature, as specified in the draft C++ standard, is implemented in MIPSpro 7.2. It is now possible for a class or class template to have templates as members. Because this feature is a pure extension, there is no flag to turn it off.

```
struct A {
  template <class T> T f(T *p) { return *p; }
};

int main() {
  int *p = new int(1);
  A a;
  int i = a.f(p); // Invokes A::f(int *)
}
```

Partial Specialization of Class Templates

The partial specialization of class templates is supported in MIPSpro 7.2 C++. A *partial specialization* is a version of a class template that is specialized for a subset of the templatized class. When an instantiation of the class is required, the front end selects the partial specialization that most closely matches the argument list. Lines `#2` and `#3` in the following example are partial specializations of line `#1`:

```
template <class T, class U> struct A { int i; A() : i(1) {} };        // #1
template <class T, class U> struct A<T*, U> { int i; A() : i(2) {} }; // #2
template <class T> struct A<T*, T>   { int i; A() : i(3) {} };        // #3

A<int, char> a1;  // Uses #1
A<int*, char> a2; // Uses #2
A<int*, int> a3;  // Uses #3

int main() {
  printf("%d %d %d\n", a1.i, a2.i, a3.i);  // Output is 1 2 3
}
```

New Specialization Syntax

MIPSpro 7.2 implements the new specialization syntax, **template<>**. The prefix **template<>** indicates that the specialization does not require a template parameter. Some examples follow:

```
template <class T> struct A {
  template <class T2> void f(T);
  template <class T2> struct B {};
};

// Definition of member template A<T>::f
template <class T> template <class T2> void A<T>::f(T2) {}

// Explicit specialization of template A<int>::f
template <> template <class T2> void A<int>::f(T2) {}

// Explicit specialization of instance of template A<int>::f(double)
template<> template<> void A<int>::f(double){}

// Explicit specialization of A<char>
template <> struct A<char> {}

template <class T> void glob_func(T);

// Explicit specialization of glob_func(int)
template <> void glob_func(int) {}
```

To maintain compatibility, the old-style specialization, which does not use **template<>**, is also supported in MIPSpro 7.2.

New Keywords

Four keywords have been introduced with MIPSpro 7.2 C++:

explicit Prevents a constructor from being implicitly invoked. Disabled by the option **-LANG:explicit=off**.

mutable A storage specifier that says an object member can never be **const**. Disabled by the option **-LANG:mutable=off**.

namespace See "Namespaces" on page 6. Disabled by the option **-LANG:namespace=off**.

typename Allows an entity to be declared and treated as a type. Usually used with templates. Disabled by the option **-LANG:typename=off**.

See the section "Error 1040" on page 70 for porting problems related to these keywords.

Other Features and ABI Changes in the MIPSpro C++ Compilers

Both the N32 and N64 compilers have features that were introduced in MIPSpro 6.2 and MIPSpro 7.0. Of these features, only exception handling is available in the O32 compiler; you have to protect your use of these features with **#ifdef**s if you want your code to be portable across all Silicon Graphics C++ compilers.

There is a built in macro, **__EDG_ABI_COMPATIBILITY_VERSION**, which is undefined in the O32 compiler, but set to the integer value 229 in the N64 and N32 compilers. Either you can use this macro to protect your use of the language features of the N32 and N64 compilers, or you can create your own **#ifdef**.

Operators new[] and delete[]

The N32 and N64 compilers implement the array variants of operators **new** and **delete** from the draft C++ standard. All calls to allocate and deallocate arrays of objects using **new Classname[n]** and **delete[] Classname** go through operators **new[]** and **delete[]** respectively, instead of the operators **new** and **delete**. This implies the following:

- If you override the global **::operator new()**, you probably don't have to do anything else. The default library **::operator new[]()** simply 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)
  ```

  that is bound to calls of the form:

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

  If you do not do this, the compiler issues an error that an appropriate **operator new[]()** has not been 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[]()**.

Note: If you do not protect these declarations under a macro like the one described in the introduction to this section, you will get compiler errors with the O32 compiler.

Built-in bool Type

There is a built-in **bool** type in the N64 and N32 MIPSpro compilers (but not in the O32 compiler). The keywords **true** and **false** are now keywords when **bool** is supported as a built-in type, having values that are the **bool** equivalent of 1 and 0 respectively.

To take advantage of this type in a fashion that is portable between O32 and the MIPSpro compilers, you can declare a **bool** type for O32 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 predefined to be 1 when the **bool** keyword is supported.

Note: The **-LANG:bool=off** option in the N32 and N64 compilers can be used to disable the built-in **bool**, **true**, and **false** keywords (in case you have already used any of these identifiers in your program), making it behave like the O32 compiler in this regard.

Built-in wchar_t Type

The type **wchar_t** is a keyword and built-in type in the two MIPSpro compilers. It is analogous to the **wchar_t** type defined in *stddef.h*. In fact, this file can be safely included into an N32 or N64 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 the O32 and the two MIPSpro compilers.

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

Because the built-in **wchar_t** is considered a distinct type, not a synonym for **int** or **long**, there is the potential for problems. For example, consider what happens 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**:

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

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

// The fix 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, the O32 compiler will complain that **foo(long)** has already been declared. This happens because in O32 **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 and N64 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. This is necessary to support the two-argument form of **operator delete()**.

Exception Handling

Exception handling is on by default in the MIPSpro 7.x compilers in the N32 and N64 modes; it can be turned off by using the **-LANG:exceptions=off** option. It is also supported in the O32 compiler by using the **-exceptions** flag. The exception handling constructs of C++ permit you 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 that can be handled by displaying an appropriate message and possibly increasing the allocatable heap size.

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

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 raised by **throw**. 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** and **catch** block combination is as follows:

```
try {
  ...
}

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

// Catch exceptions of float thrown in the try block above
catch(float ) {
  ...
}

// Catch ALL exceptions
catch(...) {
  ...
}
```

A **try** block can be followed by zero or more **catch** blocks. If no exceptions are raised during the execution of a **try** block, control shifts to immediately after the last **catch** block. If an exception is raised, that is to say an object is thrown, during the execution of a **try** block, then the **catch** block whose type specification matches the type of the thrown object is chosen and control transfers to that handler. The block headed by the `catch(...)` statement catches all exceptions.

Run-time Type Identification

This feature is available only in 7.0 and later versions of the two MIPSpro 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 be known only at run time and it becomes necessary to provide some form of type safety. Specifically, given a pointer to an object of a base class, you may wish to determine whether it is, in fact, a pointer to an object of a specific derived class of that base class. Consider the following example:

Example 1-2 Run-time 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 by using the **dynamic-cast** facility of C++:

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

If the pointer *pb* is actually pointing to an object of the derived class rather than of the base class, the **dynamic_cast** operation returns the pointer. 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 standard library type called **type_info**, which represents the type name of its argument.

Other ABI Changes

There are three other ABI changes of note:

- The object layout has been modified somewhat between O32 and N32, specifically for classes that have virtual base classes inherited along more than one path. This may cause you some difficulties if your code depends on the exact layout of such objects.
- Name mangling for O32 is slightly different from that of the N32 and N64 compilers: The function designator *F* (as in *foo__FUi*) in O32 becomes *G* in the mangled names of the two MIPSpro compilers (as in*foo__GUi*).

  If your assembly, C or Fortran code has calls to mangled C++ names, or if you do dynamic name lookups using **dlsym** on mangled C++ names, you may be affected.
- Virtual function tables, which are internal to the implementation, have been expanded in the two MIPSpro compilers. *Subobjects* (single occurrences of a base class) are no longer limited to 32KB and 32,000 virtual functions.

  The limits are now 4GB subobject sizes (both in N32 and N64) and 4 billion virtual functions. The subobject size is not even an issue unless the class is larger than 4GB and is a non-rightmost base class in a multiple-inheritance situation; for single-inheritance, the base class size should never be an issue.

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 compiled successfully with cfront may not compile under the Silicon Graphics C++ environment, even in cfront-compatibility mode. You must compile with *CC -32 -use_cfront* 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 operators **::new** and **::delete**. All N32 and N64 programs also link with the *libCsup.so* library, which provides exception handling and Run-time Type Information support; this library is also used for O32 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
```

For more information on the *complex* and *iostream* libraries, see *The C++ Programming Language* referenced in "C++ Language References" on page xiv.

Debugging

You can debug your C++ programs with *dbx*—a source-level debugger for C, C++, Fortran, and assembly language, or with the ProDev WorkShop Debugger—a graphical, interactive, source-level debugging tool. For more information on *dbx*, see the *dbx User's Guide*. For additional information on the WorkShop Debugger, see the *Developer Magic: Debugger User's Guide*.

Chapter 2

Compiling, Linking, and Running C++ Programs

This chapter contains two major sections:

- "Compiling and Linking" on page 20 describes the Silicon Graphics C++ compilation process and how to compile and link C++ programs. It consists of the following sections:
  - "Translators and Drivers" on page 20
  - "Compilation" on page 21
  - "Sample Command Lines" on page 23
  - "Multi-Language Programs" on page 24
- "Object File Tools" on page 24 briefly summarizes the capabilities of the tools that provide symbol and other information on object files.

Compiling and Linking

This section discusses C++ compiling and linking for the N32, N64, and O32 compilers.

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 this and the following section. The *CC* command, see the CC(1) reference page, invokes the driver that controls compilation of your C++ source files. The syntax is as follows:

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

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

CC compiles with many of the same options as *cc*, the C language driver. See the CC(1) and cc(1) reference pages for more information about available options. Also of interest is the *MIPSpro Compiling and Performance Tuning Guide*, which discusses the following tools for optimization:

- **-O0**, **-O1**, **-O2**, **-O3**, and **-Ofast** optimization options
- Interprocedural analysis (**-IPA:...**)
- Loop nest optimization (**-LNO:...**)
- Floating point and miscellaneous optimizations (**-OPT:...**)
- Precompiled headers

Compilation

The two compilation processes in Figure 2-1 show what transformations a C++ source file, *foo.C*, undergoes with the MIPSpro (N32 and N64) compilers and the ucode (O32) compiler. On the left is the MIPSpro compilation process, invoked using either **-n32** or **-64** mode:

```
CC -n32 –o foo foo.C
CC -64 –o foo foo.C
```

On the right is the ucode compilation process, invoked with **-o32**:

```
CC -o32 –o foo foo.C
```

The following steps further describe the compilation stages in Figure 2-1:

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

   This stage may also produce a prelink (*.ii*) file, which contains information about template instantiations.
4. The back end generates optimized object code (*foo.o*).

   Note: To stop the compilation at this stage, instead of the command line

   `CC` *mode* `-o foo foo.C`

   use

   `CC` *mode* `–c foo.C`

   This command produces object code, *foo.o*, that is suitable for later linking.

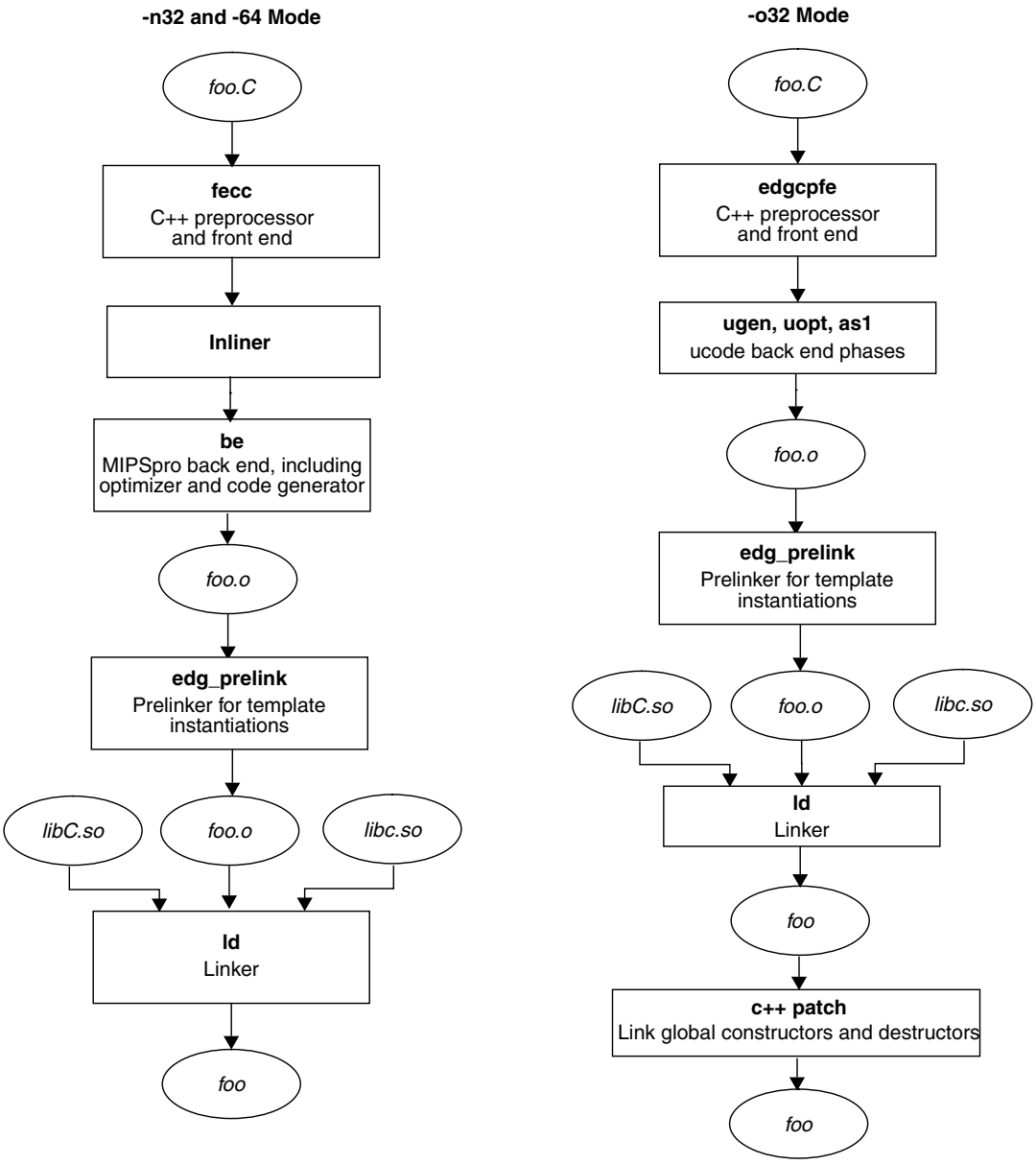

Figure 2-1 The MIPSpro and ucode C++ Compilation Processes

5. edg_prelink processes the *.ii* files associated with the objects that will be linked together. It then recompiles sources to force template instantiation.
6. The object files are sent to the linker, *ld*, (see the ld(1) reference pages) 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 are needed to produce the executable.
7. In **-o32** mode, 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()**, and the destructors need to be called when the program exits. *c++patch* modifies the executable, *a.out*, to ensure that these constructors and destructors get called.

Sample Command Lines

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

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

  ```
  CC -c program
  ```

- To compile with full warnings about questionable constructs, use the following command line:

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

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

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

- To compile in N64 mode with cfront compatibility enabled, use the following command line:

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

- To compile in O32 mode with cfront compatibility disabled, use the following command line:

  ```
  CC -o32 +p program1 program2 ...
  ```

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 do as follows:

1. Compile each program module separately with the appropriate driver.
2. Link them together 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 three compilers 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. In fact, if any object file is generated by C++, it is best to use *CC* to link. Except for C, you must explicitly specify the link libraries for the other languages with the **–l** option. For example, to link the C++ main module with the Fortran submodule, you use the following command line:

```
CC -o almostall main.o module1.o -lftn -lm
```

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

Object File Tools

The following object file tools are of special interest to the C++ programmer:

nm	This tool can print symbol table information for object and archive files.
c++filt	This tool, specifically for C++, translates the internally coded (mangled) names generated by the C++ translator into names more easily recognized by the programmer. You can pipe the output of *stdump* or *nm* into *c++filt*, which 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 program to output a readable form of a mangled name. This is useful for writing your own tool for processing the output of *nm*, for example. You need to declare

```
char * demangle(char *);
```

in your program, and link with the library using 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 of an ELF-format object file, including the symbol table and header information. See the elfdump(1) reference pages for more information.

stdump The *stdump* tool outputs a file of intermediate-code symbolic information to standard output for O32 executables only. See the stdump(1) reference pages for more information.

dwarfdump The *dwarfdump* tool outputs a file of intermediate-code symbolic information to standard output for **-n32** and **-64** compilations. See the dwarfdump(1) reference pages for more information.

For more complete information on the object file tools available to you, consult the *MIPSpro Compiling and Performance Tuning Guide.*

Chapter 3

C++ Dialect Support

This chapter describes the C++ language implemented by the two MIPSpro C++ compilers. The ucode compiler accepts an older version of the C++ language, which is not covered.

This chapter contains the following major sections:

- "About the Front End" on page 28 contains background information on the MIPSpro C++ front end, fecc.
- "New Language Features" on page 29 contains a list of MIPSpro C++ compiler features that are not in the *Annotated C++ Reference Manual* (ARM), but are in the X3J16/WG21 Working Paper.
- "Non-implemented Language Features" on page 31 contains a list of features not supported by the MIPSpro C++ compilers that are not in the ARM, but are in the X3J16/WG21 Working Paper.
- "Anachronisms Accepted" on page 32 contains a list of the anachronisms that are supported in the MIPSpro compilers when the **-anach** option is enabled.
- "Extensions Accepted in Default Mode" on page 33 contains a list of the extensions that are accepted by the MIPSpro compilers by default.
- "Extensions Accepted in Cfront-Compatibility Mode" on page 34 contains a list of extensions accepted by the MIPSpro compilers in cfront-compatibility mode.
- "Cfront Compatibility Restrictions" on page 38 contains a list of constructs that cfront supports but the MIPSpro compilers reject.

About the Front End

The front end, fecc, accepts the C++ language as defined by *The Annotated C++ Reference Manual* (ARM) including templates, exceptions, and the anachronisms discussed in this chapter. This language is augmented by extensions from the X3J16/WG21 Working Paper of the standards committee.

The command line option **-anach** enables anachronisms from older C++ implementations. By default, anachronisms are disabled. See "Anachronisms Accepted" on page 32 for details.

By default, the front end accepts certain extensions to the C++ language; these extensions are flagged as warnings if you use the **-ansiW** option, and as errors if you use the **-ansiE** option. See "Extensions Accepted in Default Mode" on page 33 for details.

The front end also has a cfront-compatibility mode (enabled by the **-cfront** option), which duplicates a number of features and bugs of cfront, an older C++ compiler front end that accepted output from a C preprocessor and gave input to a C compiler. Complete compatibility is not guaranteed or intended—the mode is there to allow programmers who have used cfront features to continue to compile their existing code. By default, the front end does not support cfront compatibility. See "Extensions Accepted in Cfront-Compatibility Mode" on page 34 and "Cfront Compatibility Restrictions" on page 38 for details.

New Language Features

The following features, not in the ARM but in the X3J16/WG21 Working Paper, are accepted by the MIPSpro C++ compilers.

- 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.
- 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.
- Run-time 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.
- Definition of nested classes outside of the enclosing class is allowed.
- **mutable** is accepted on non-static data member declarations.
- Namespaces are implemented, including **using** declarations and directives. Access declarations are broadened to match the corresponding **using** declarations.
- Explicit instantiation of templates is implemented.
- The **typename** keyword is recognized.
- **explicit** is accepted to declare non-converting constructors.
- The scope of a variable declared in the *for-init-statement* of a **for** loop is the scope of the loop, not the surrounding scope.
- Member templates are implemented.
- The new specialization syntax (using **template <>**) is implemented.
- *Cv-qualifiers* (*cv* stands for **const volatile**) are retained on rvalues (in particular, on function return values).
- The distinction between trivial and non-trivial constructors has been implemented, as has the distinction between POD (point of definition) constructs, such as, C-style **struct**s, and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions.)
- A **typedef** name may be used in an explicit destructor call.
- Placement **delete** is implemented.
- An array allocated via a placement **new** can be deallocated via **delete**.
- **enum** types are considered to be non-integral types.
- Partial specialization of class templates is implemented.

Non-implemented Language Features

The following features which are not in the ARM but are in the X3J16/WG21 Working Paper, are not accepted by the MIPSpro C++ compilers:

- Covariant return types on overriding virtual functions are not supported.
- **extern inline** functions are not supported.
- Digraphs are not recognized.
- Operator keywords (for example, **and**, and **bitand**) are not recognized.
- It is not possible to overload operators using functions that take **enum** types and no class types.
- The new lookup rules for member references of the form **x.A::B** and **p->A::B** are not yet implemented.
- **enum** types cannot contain values larger than can be contained in an **int**.
- **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.
- Name binding in templates in the style of N0288/93-0081 is not implemented.
- 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.
- Class name injection is not implemented.
- Overloading of function templates (partial specialization) is not implemented.
- Putting a **try** and **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();
};
```

- The preprocessing symbol **c_plusplus** is defined in addition to the standard **__cplusplus**.

- 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 is 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 actually does 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-compatibility 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.

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

  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 non-standard 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 non-standard 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. Version 2.1 of cfront 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-compatibility 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-compatibility mode it is reduced to a warning. For example:

```
class A {
  ~A();
};
class B : public A {
  ~B();
};
B::~B(){}    // Error except in cfront-compatibility 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 statement `A(x2);` is also misinterpreted by cfront. It should be interpreted as the declaration of an object named **x2**, but in cfront-compatibility mode is interpreted as a function style cast of **x2** to the type **A**.

  Similarly, the statement

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

  declares a function named **xyz**, that takes a parameter of type "function taking no arguments and returning an **int**." In cfront-compatibility 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;              // Not an error in cfront-compatibility mode
```

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

```
short short int i;  // No warning given in cfront-compatibility 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 new 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-compatibility 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-compatibility mode
```

Cfront Compatibility Restrictions

Even when you specify the **-cfront** option, the N32, N64, and O32 C++ compilers are not completely backwards-compatible with cfront. The N32, N64, and O32 compilers reject the following source constructs that cfront ignores:

- Assignment to **this** in constructors and destructors is not allowed. (O32 generates a warning.)
- If a C++ comment line (//) is terminated with a backslash, the MIPSpro compilers (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 pass a pointer to **volatile** data to a function that is expecting a pointer to non-**volatile** data.
- The MIPSpro compilers do not disambiguate between overloaded functions with a **char*** and **long** parameter, respectively, when called with an expression that is a zero cast to a **char** type.

- You may not use redundant type specifiers.
- When in a conditional expression, the MIPSpro compilers do 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" is treated as an **int**.
- The MIPSpro compilers *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 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

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 56 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 59 describes the language features for templates supported in the Silicon Graphics C++ environment, but not in cfront.

For general information about the Standard Template Library (STL)—a library of container classes, algorithms, and iterators for generic programming, see the html document available at http://www.sgi.com/Technology/STL/. The MIPSpro 7.2 C++ Release Notes have information about the particular version of the STL being shipped with the 7.2 release.

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 either 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. However, as explained in "Explicit Instantiation" on page 46, the use of the **-ptused** option is being deprecated.

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 45.
- 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 sees 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 notices that too and removes 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 all the conditions below are 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 following conditions are true:

- 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** option 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 Template Instantiation" on page 54 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 can use command-line options to control the instantiation behavior of the compiler. These options are divided into four 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**. (Automatic template instantiation should make the use of **-ptused** and **-ptall** unnecessary in most cases.)
- **-prelink** (the default) and **-no_prelink**.
- **-auto_include** and **-no_auto_include**.
- **-ptv**.

The command line options are discussed 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.

Note: The use of the **-ptused** option is being deprecated in the MIPSpro 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.

Note: The use of the **-ptall** option is being deprecated in the MIPSpro 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 Disables automatic instantiation. 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 45.)

-no_auto_include
Disables implicit inclusion of template implementation files. (See "Implicit Inclusion" on page 45.)

-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 typical combinations of command line instantiation options, along with an explanation of what these combinations 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. Dynamically initialized static data members are also handled correctly (by using a runtime guard to prevent duplicate initialization of such members).

  Note: The use of this option is being deprecated.

-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, Silicon Graphics recommends that you run the prelinker on the object files before archiving them. However, there are cases where you 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 entities 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 implement a stack template class containing various member functions. You may choose to provide instantiated versions of these functions for various common types (such as, **int** and **float**) 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 and know that all of your referenced template entities have already been pre-instantiated in a library such as one 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.

Pragmas for Template Instantiation

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

#pragma instantiate
: Causes a specified entity to be instantiated.

#pragma 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.

#pragma 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 pragmas may be any of the following:

- 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 either 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 any of the following:

- a compiler-generated function
- an inline function
- 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 51 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. You can have the prelinker automatically instantiate all needed templates in either of two ways:

- Build a shared library with a command like

  ```
  CC -n32 -shared ...
  ```

- Build an archive with a command like

  ```
  CC -ar ...
  ```

Limitations on Template Instantiation

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

Unselected Template Specializations in Archives

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 is 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 correctly selects specializations from dynamic shared objects.

Undetected Link-Time Changes in Template Implementation Files

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 a *Makefile* 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*.

Prelinker Cannot Recompile Renamed Files

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 53.)
- 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 are not 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 link it separately, instead of compiling and linking in a single step. Another method is to create an empty dummy file, and then compile and 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 repositories are not 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 58 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 51.

-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.

Users of cfront do this to build an archive or library that 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 53 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 all uninstantiated template functions can be instantiated in the object files being linked into the application, the prelinker does 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 **class**es, **typedef**s, and **enum**s in class templates, including variant **typedef**s and **enum**s. (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. The code calls would be 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;
```

  This code causes cfront to generate an error.

- 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) {}
```

- *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.

Appendix A

C and C++ Pragma Directives

This appendix discusses the **#pragma** directives available for the Silicon Graphics C and C++ environment. It contains the following sections:

- "Common Recognized Pragmas" on page 63 discusses some of the most frequently used pragmas.
- "Other References on Pragmas" on page 68 lists references for those pragmas not covered in this appendix, particularly pragmas intended for multiprocessing.

Common Recognized Pragmas

The MIPSpro compilers recognize many commonly used pragmas. The following ones are discussed in this section:

- "#pragma can_instantiate" on page 64
- "#pragma do_not_instantiate" on page 64
- "#pragma hdrstop" on page 64
- "#pragma instantiate" on page 64
- "#pragma int_to_unsigned" on page 65
- "#pragma intrinsic" on page 65
- "#pragma no side effects" on page 65
- "#pragma once" on page 66
- "#pragma pack" on page 66
- "#pragma weak" on page 67

#pragma can_instantiate

#pragma can_instantiate *declaration*

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.

#pragma do_not_instantiate

#pragma do_not_instantiate *declaration*

The opposite of **#pragma instantiate**. If the compiler sees this pragma, it does not instantiate the specific declaration in the compilation unit, even if there is an instance of it.

#pragma hdrstop

#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 *MIPSpro Compiling and Performance Tuning Guide* for details on the precompiled header mechanism.

#pragma instantiate

#pragma instantiate *declaration*

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

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

The declaration needs to be a complete declaration of either a function or a static data member. You specify it exactly as you would specify a specialization of the template. See also “Pragmas for Template Instantiation” on page 51

#pragma int_to_unsigned

#pragma int_to_unsigned *identifier*

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

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

Note: Although **#pragma int_to_unsigned** is silently recognized by the MIPSpro C++ compilers, it has no effect.

#pragma intrinsic

#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 no side effects

#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 once

#pragma once In **-n32** and **–64** modes, this pragma ensures idempotent **#include** files (that is, **#include** files are included at most once in a compilation unit).

Silicon Graphics recommends enclosing the contents of **#include** files, such as *afile.h*, with an **#ifdef** directive similar to:

```
#ifndef afile_INCLUDED
#define afile_INCLUDED
<contents of afile.h>
#endif
```

Note: This pragma has no effect in **–o32** mode.

#pragma pack

#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 the default alignment for any subsequent **struct** definitions.

A **struct** type defined in the scope of a **#pragma pack(*n*)** has an alignment of at most *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.

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

#pragma weak

#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 namespace. 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 namespace.

Note: **#pragma weak** is not supported in O32.

Other References on Pragmas

The following references also discuss pragmas. They cover many pragmas that are not found in this appendix:

- For a more complete treatment of pragmas, see *MIPSpro C and C++ Pragmas*, which is available through the Silicon Graphics Technical Publications Library at http://techpubs.sgi.com/library/.
- For information about pragmas for automatic and manual parallelization, see the *MIPSpro Automatic Parallelizer Programmer's Guide* and the *C Language Reference Manual*, respectively.
- For information about pragmas for data distribution on Origin2000™ systems, see either the *MIPSpro Fortran 77 Programmer's Guide* or *MIPSpro C and C++ Pragmas*. These pragmas include the following:
  - **#pragma distribute**
  - **#pragma redistribute**
  - **#pragma distribute_reshape**

Appendix B

MIPSpro 7.2 C++ Porting Errors

As the MIPSpro C++ compilers evolve to meet the requirements of the draft standard, incompatibilities develop between the latest compilers and existing code. The incompatibilities are manifested by code that no longer compiles, or code whose run-time behavior changes. In either case, error messages can be generated during compilation.

This appendix describes error messages you may receive when you port code to the MIPSpro 7.2 C++ compilers from the 7.1 versions. In each case, the error message and a description of the cause are given. Where there is a solution, it is provided. The following errors are included:

- "Error 1040" on page 70: `expected an identifier`
- "Error 1140" on page 71: `an enum cannot be used to initialize a pointer`
- "Error 1164" on page 71: `argument type of enum is incompatible with parameter of type void *`
- "Error 1168" on page 71: `invalid type conversion`
- "Error 1220" on page 72: `function "operator new(size_t, void *)" has already been defined`
- "Error 1320" on page 73: `nonmember operator requires a parameter with class type`
- "Error 1324" on page 73: `more than one operator ? matches these operands`
- "STL Error: Error 1278" on page 74: `no instance of overloaded function ...`
- "Warning 1306" on page 76: `class "..." has no copy assignment operator`

Error 1040

```
error(1040): expected an identifier
```

This error occurs because of an improperly used reserved word. As described in "New Features of the MIPSpro 7.2 C++ Compilers" on page 6, there are four new reserved words:

- **explicit**
- **mutable**
- **namespace**
- **typename**

If you are unable to modify code that uses these reserved words, as in the case of third party headers, you can use the following options, in any combination, to prevent the compiler from recognizing the words as reserved:

- **-LANG:explicit=off** (undefine the **_EXPLICIT_IS_KEYWORD** macro)
- **-LANG:mutable=off** (undefine the **_MUTABLE_IS_KEYWORD** macro)
- **-LANG:namespace=off**
- **-LANG:typename=off**

Using the first two options makes the two related macros, which are normally defined, undefined. The macros can be used in source file **#ifdef** statements for conditional compilation based on the features that are enabled.

Error 1140

```
error(1140): an enum cannot be used to initialize a pointer
```

An **enum** in C++ is not a numeric type, and there is no implicit conversion to a pointer. The MIPSpro 7.1 C++ compiler did not diagnose this error, but the 7.2 C++ compiler does. The following is an example of faulty code and the error message that results:

```
enum fruit {orange};
void* p = orange; // This is an error
```

```
"foo.c", line 2: error(1140): a value of type "fruit" cannot be used to
          initialize an entity of type "void *"
     void* p = orange;
```

To fix this problem, explicitly cast the **enum** to an **int**, as shown

```
enum fruit {orange};
void* p = (int)orange; // This is fine
```

Error 1164

```
error(1164): argument type of enum is incompatible with parameter of
type void *
```

See “Error 1040” on page 70.

Error 1168

```
error(1168): invalid type conversion
```

See “Error 1040” on page 70.

Error 1220

```
error(1220): function "operator new(size_t, void *)" has already been
defined
```

Prior to MIPSpro 7.2 C++, it was possible to define your own version of placement **operator new**. A placement version of **operator new** was defined in *new.h* as follows:

```
void *operator new(size_t, void* p);
```

The library provided this implementation:

```
void *operator new(size_t, void* p) { return p; }
```

The C++ standard now states that the standard version of placement **operator new** cannot be preempted. Thus an inline version of placement **operator new** can be provided in *new.h*:

```
inline void *operator new(size_t, void* p) { return p; }
```

As a consequence, if you explicitly define your own version of placement **operator new** (inline or not), you will get an error, as this example shows:

```
#include <new.h>
inline void *operator new(size_t, void* p) { return p; }
```

```
"foo.c", line 2: error(1220): function "operator new(size_t, void *)" has
          already been defined
     inline void *operator new(size_t, void* p) { return p; }
```

Note: You are permitted to define versions of placement **operator new** with parameter profiles that differ from those the standard version.

The contents of this section also apply to **operator new[]**.

Error 1320

```
error(1320): nonmember operator requires a parameter with class type
```

Nonmember operators have always required at least one class (or reference to class) parameter. C++ compilers prior to the 7.2 release have occasionally failed to diagnose this problem, particularly in template functions. MIPSpro 7.2 C++ always gives a compile-time error for code such as this:

```
template <class T>
class Ptr{
};

template <class T>
int operator!=(Ptr<T>* a, Ptr<T>* b); // Should be an error
```

The solution is to remove every function like this. Removing these functions cannot change the program's behavior, because they could not have been invoked by earlier compilers.

Error 1324

```
error(1324): more than one operator ? matches these operands
```

This error pertains to conditional expressions of the form *condition* **?** *statement* **:** *statement* where each *statement* is a different type. For example:

```
struct A {
   A();
   A(int);
   operator int();
};

void foo(int i,int j) {
   i ? A() : j; // Ambiguous in 7.2 -n32
}
```

```
"foo.c", line 8: error(1324): more than one operator "?" matches these operands:
            built-in operator "expression ? class : class"
            built-in operator "expression ? arithmetic : arithmetic"
            operand types are: A : int
            Recent changes to Ansi C++ draft standard may have
            made this expression ambiguous. Please rewrite
            the expression to remove the ambiguity. To get the old behavior
            use "-Wf,-old_conditional_operator_rules" flag.
    i ? A() : j; // Ambiguous in 7.2 -n32
```

A compiler can determine the return type of the statement `i ? A():j;` in two ways:

- It can convert **A()** to an **int**, making the result of the expression type **int**.
- It can convert the **int** to an **A**, making the result type **A**.

Before the MIPSpro 7.2 release, the compilers preferred converting **A()** to an **int**. The draft standard now makes this construct ambiguous. The recommended way of making this code work with all compilers is to use an explicit cast to make the result type unambiguous. What the example probably intends is the following:

```
i ? (int)A() : j;
```

The alternate choice is this:

```
i ? A() : (A)j
```

If you cannot change your source code to use an explicit cast, use the **-Wf,-old_conditional_operator_rules** flag to revert to the old rules.

STL Error: Error 1278

```
STL Error: error(1278): no instance of overloaded function ...
```

The following Standard Template Library (STL) code produces an error that arises because the STL uses member templates and MIPSpro 7.2 C++ now supports member templates.

```
#include <vector.h>
vector<int> v(5,0);
```

This is the error message:

```
"/hosts/alliant/72_test/LATEST/usr/include/CC/vector.h", line 99: error(1278):
          no instance of overloaded function "iterator_category" matches the
          argument list
            argument types are: (int)
          range_initialize(first, last, iterator_category(first));
                                        ^
          detected during instantiation of
                  "vector<int, alloc>::vector(int, int)" at line 2 of "foo.c"

1 error detected in the compilation of "foo.c".
```

The intent of the statement `vector<int> v(5,0);` is to create a vector with five elements, each of which is initialized to zero. The problem is that **vector<T>** has two different constructors, each with its own signature:

- One constructs a vector with *N* elements, each having an initial value *x*

  ```
  vector<T>::vector(size_t N, const T& x);
  ```

- One constructs a vector with a range of [*first*, *last*)

  ```
  // This is a member template
  template <class Iterator>
  vector<T>::vector(Iterator first, Iterator last);
  ```

The compiler sees that the two arguments of **vector<int> v(5,0)** are of the same type, so they match the constructor that accepts a range. Because the first element is an **int**, not a **size_t**, it is not an exact match for the first constructor. When the compiler tries to use the second constructor, it issues the error message because an **int** cannot be used as an iterator.

The code can be fixed by casting the first argument to **size_t**.

```
vector<int> v((size_t)5,0);
```

If code you are porting from MIPSpro C++ 7.1 to 7.2 uses the STL, this problem can occur whenever the code uses the two-argument constructor for **vector<***integer_type***>** where *integer_type* is any integral type. This is a bug in the draft C++ standard, and the standards committee is investigating solutions.

Warning 1306

```
warning(1306): class "..." has no copy assignment operator
```

This error indicates a change in the run-time behavior of copy assignment operator calls. It affects code with classes that have assignment operators that accept arguments of a different class type.

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

main()
{
   B b1, b2;
   b1 = b2;
   return 0;
}
```

With the 7.1 compilers, the statement `b1 = b1;` generates a call to the conversion operator function, **operator A& () const**, to convert an object of type **B** to type **A**. Then a call to the function **B& operator= (const A&)** copies the **A** object.

This behavior is incorrect because the function **B& operator= (const A&)** is not a copy assignment operator. If it were a copy assignment operator, it would take exactly one of the following parameters:

- **B&**
- **const B&**
- **volatile B&**
- **const volatile B&**

With the 7.2 compilers, the front end implicitly generates a copy assignment operator, **B::operator=(const B&)**, that performs a bit-wise copy. This may cause completely different run-time behavior, and the compiler gives this warning:

```
"t.c", line 12: warning(3149): class "B" has no copy assignment operator. A
          copy assignment operator of the form: function
          "B::operator=(const B &)" has been implicitly defined. The
          implicitly defined copy assignment operator will perform a bitwise
          copy. Previous compiler versions may have used assignment operator
          functions defined in this class with conversion operators as copy
          assignment operator. This change may lead to different runtime
          behavior. Define a copy assignment operator if necessary.
     b1 = b2;
```

The suggested solution to the problem is to define a copy assignment operator that performs the copy. For the previous case, one could add this member function in **B**:

```
B& operator= (const B& arg) {
  *this = ( const A &) arg; // Convert the argument to a 'A' object and
                            // then call the operator= function.
}
```

Glossary

Terms shown in *italics* indicate glossary items, document titles, or variables.

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

application binary interface (ABI)
It defines a system interface for executing compiled programs.

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 *member function*.

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.

instruction set architecture (ISA)
The set of instructions recognized by a processor.

mangle
Encoding a function name to support overloading.

member function
C++ function that is a member of a class or **struct** data type.

members

Either or both of *data* and *member function*s belonging to a class (class members).

namespace

In C++, a scope used for logical grouping.

partial specialization

In C++, a version of a class template that is specialized for a subset of the more general class.

private

In C++, access to the class member is restricted to following:

- the class in which it is defined
- friend classes
- friend functions

protected

In C++, access to the class member is restricted to the following:

- the class in which it is defined
- all derived classes
- friend classes
- friend functions

public

In C++, access is open to any 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 can_instantiate, 64
#pragma do_not_instantiate, 64
#pragma hdrstop, 64
#pragma instantiate, 64
#pragma int_to_unsigned identifier, 65
#pragma intrinsic, 65
#pragma no side effects, 65
#pragma once, 66
#pragma pack, 66
#pragma weak, 67

A

ABI, 64-bit changes, 10
ABI changes, 10
ABI changes, other, 17
accepted anachronisms, 32
anachronisms, accepted, 32
assignment to this, 13

B

bool, 12
built-in bool type, 12
built-in w_char_t type, 12

C

C++ compilers
  6.0 versions, 2
  64 vs. 32 bit, 4
C++ environment, 2
c++patch, 23
catch, 14
Cfront
  compatibility with Delta/C++, 17
cfront, compatibility restrictions, 38
cfront compiler, 2
cfront template transition, 56
changes, 64-bit ABI, 10
command lines
  samples, 23
compatibility restrictions, cfront, 38
compiler
  cfront, 2
  N32, 2
  N64, 2
  O32, 2
  ucode, 2
compilers
  64 vs. 32 bit, 4
compiling, 20, 21
compiling, and linking, 20
complex arithmetic library, 18
contents of guide, xiii
conventions, font, for manual, xvi

D

debugging, 18
delete, operator, 11
Delta/C++
  Cfront compatibility with, 17
dialect support, C++, 27
documentation, recommended reading, xiv

E

exception handling, 14
  catch, 14
  throw, 14
  try, 14
exctensions
  accepted default, 33
extensions
  accepted cfront, 34
  cfront accepted, 34
  default accepted, 33

F

features
  new language, 29
  non-implemented, 31
features, new, 10
font conventions, for manual, xvi
front end
  about, 28

G

global constructors, 23
guide contents, xiii

H

handling, exception, 14

I

implicit inclusion, 45
inclusion, implicit, 45
instantiation
  automatic, details of, 44
  automatic method of, 43
  requirements, 43
instantiation, command-line options, 47
instantiation, template, 46
iostream library, 18

L

language, new features, 29
ld, 23
libraries, 18, 24
link editor, 23
linking, 20, 24
linking, and compiling, 20
link libraries, 24
loader, 23

M

member templates, 8
multi-language programs, 24

N

N32 compiler, 2
N32/N64 compilers, 2
N64 compiler, 2
namespace, 6
NCC
  mapping template options from cfront, 56
new, operator, 11

O

O32 compiler, 2
object files, 20
  linking, 24
  tools, 24
operators new and delete, 11

P

partial specialization, 8
pragmas, 67
  for template instantiation, 51
ptrepository, object files in, 58

R

repositories, multiple template, 58
RTTI, 16
runtime type identification, 16

S

scope
  namespace, 6
shared libraries, building, 53
source file, suffix, 21
specialization, 53

T

template, 9
template instantiation
  archives, 53
  shared libraries, 53
templates
  automatic instantiation, 43
  command-line instantiation, 47
  instantiation, 46, 51
  instantiation examples, 49
  introduction to instantiation, 42
  language support, 59
  mapping cfront to NCC options, 56
  multiple repositories, 58
  object files in cfronts ptrepository, 58
  restrictions, 54
  specialization, 53
  transitioning from cfront, 56
this, assignment to, 13
throw, 14
tools, object files, 24
try, 14
type identification, runtime, 16

U

ucode compiler, 2

W

wchar_t, 12
weak_signal=strong_symbol, 67

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 007-0704-120.

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: 650-965-0964
- 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