C++ Programmer's Guide

Document Number 007–0704–130

St. Peter’s Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.

Copyright © 1995, 1999 Silicon Graphics, Inc. All Rights Reserved. This document or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Silicon Graphics, Inc.

LIMITED AND RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the Government is subject to restrictions as set forth in the Rights in Data clause at FAR 52.227-14 and/or in similar or successor clauses in the FAR, or in the DOD, DOE or NASA FAR Supplements. Unpublished rights reserved under the Copyright Laws of the United States. Contractor/manufacturer is Silicon Graphics, Inc., 1600 Amphitheatre Pkwy., Mountain View, CA 94043-1351.

Autotasking, CF77, CRAY, Cray Ada, CraySoft, CRAY Y-MP, CRAY-1, CRInform, CRI/*Turbo*Kiva, HSX, LibSci, MPP Apprentice, SSD, SUPERCLUSTER, UNICOS, X-MP EA, and UNICOS/mk are federally registered trademarks and Because no workstation is an island, CCI, CCMT, CF90, CFT, CFT2, CFT77, ConCurrent Maintenance Tools, COS, Cray Animation Theater, CRAY APP, CRAY C90, CRAY C90D, Cray C++ Compiling System, CrayDoc, CRAY EL, CRAY J90, CRAY J90se, CrayLink, Cray NQS, Cray/REELlibrarian, CRAY S-MP, CRAY SSD-T90, CRAY SV1, CRAY T90, CRAY T3D, CRAY T3E, CrayTutor, CRAY X-MP, CRAY XMS, CRAY-2, CSIM, CVT, Delivering the power . . ., DGauss, Docview, EMDS, GigaRing, HEXAR, IOS, ND Series Network Disk Array, Network Queuing Environment, Network Queuing Tools, OLNET, RQS, SEGLDR, SMARTE, SUPERLINK, System Maintenance and Remote Testing Environment, Trusted UNICOS, and UNICOS MAX are trademarks of Cray Research, Inc., a wholly owned subsidiary of Silicon Graphics, Inc.

Silicon Graphics, IRIX, and OCTANE are registered trademarks and O2 and the Silicon Graphics logo are trademarks of Silicon Graphics, Inc. R10000 and R12000 are trademarks or registered trademarks exclusively used under license by Silicon Graphics, Inc.

DynaText and DynaWeb are registered trademarks of Inso Corporation. MIPS, MIPSpro, R2000, R3000, R4000, R4400, R4600, and R8000 are trademarks of MIPS Technologies, Inc. OpenMP is a trademark of the OpenMP Architecture Review Board. Portions of this publication may have been derived from the OpenMP Language Application Program Interface Specification. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. X/Open is a trademark of X/Open Company Ltd. The X device is a trademark of the Open Group.

The UNICOS operating system is derived from UNIX® System V. The UNICOS operating system is also based in part on the Fourth Berkeley Software Distribution (BSD) under license from The Regents of the University of California.

New Features

C++ Programmer's Guide 007–0704–130

This revision of the *C++ Programmer's Guide* supports the 7.3 release of the MIPSpro C++ compiler. See the `cc`(1) man page for changes or additions to command line options.

The following changes and additions have been made to this document:

- The appendix, *C and C++ Pragmas*, has been removed. See the *MIPSpro C and C++ Pragmas* document for details on `#pragma` directives.
- Information about the MIPSpro Auto-Parallelizing Option (APO) has been included in Chapter 5, page 51. This information was taken from the *MIPSpro Auto-Parallelizing Option Programmer's Guide*, which will no longer be revised.
- General information about OpenMP directives supported for the MIPSpro C++ compiler is described in Section 1.4.1, page 5. See the *MIPSpro C and C++ Pragmas* manual for more information about OpenMP directives.
- The new `-LANG:std` option enforces strict ANSI conformance and uses the new C++ ANSI standard conforming library which contains a new IOSTREAM implementation (with LOCALE). See Section 1.5, page 12, for more information.

Record of Revision

Version	*Description*
7.3	April 1999 This revision supports the MIPSpro 7.3 release.

Contents

Page

Preface **vii**

Related Publications vii

Obtaining Publications vii

Conventions vii

Reader Comments ix

Understanding the Silicon Graphics C++ Environment [1] **1**

Silicon Graphics C++ Environment 1

Understanding ABIs and ISAs 2

N32, 64, and O32 Compilation 4

Feature and ABI Changes in the MIPSpro C++ Compilers 5

OpenMP API Multiprocessing Directives 5

Operators `new[ ]`and `delete[ ]` 6

Built-in `bool` Type 7

Built-in `wchar_t` Type 7

Exception Handling 8

Example 1: Exception Handling 9

Run-time Type Identification 10

Example 2: Run-time Type Identification (RTTI) 10

Other ABI Changes 11

C++ Libraries 12

Debugging 12

Compiling, Linking, and Running C++ Programs [2] **13**

Compiling and Linking 13

Page

Compilation 14
Sample Command Lines 16
Multi-Language Programs 16
Object File Tools 17

C++ Dialect Support [3] **19**

About the Compiler 19
New Language Features 20
Non-implemented Language Features 22
Anachronisms Accepted 23
Extensions Accepted in Default Mode 24
Extensions Accepted in Cfront-Compatibility Mode 25
Cfront Compatibility Restrictions 29

Using Templates [4] **31**

Template Instantiation 31
Automatic Instantiation 32
Meeting Instantiation Requirements 32
Automatic Instantiation Method 33
Details of Automatic Instantiation 33
Implicit Inclusion 35
Explicit Instantiation 35
Command Line Options for Template Instantiation 36
Command Line Instantiation Examples 38
`#pragma` Directives for Template Instantiation 40
Specialization 42
Building Shared Libraries and Archives 42
Limitations on Template Instantiation 42
Unselected Template Specializations in Archives 43

Page

Undetected Link-Time Changes in Template Implementation Files 43
Prelinker Cannot Recompile Renamed Files 44
How to Transition From Cfront 44
Mapping Template Options From Cfront to CC 44
Using Object Files From Cfront's Repository 46
What to Do If You Use Multiple Repositories 47
Template Language Support 47

The Auto-Parallelizing Option (APO) [5] **51**

Index **53**

Figures

Figure 1. Silicon Graphics C++ Environment 2
Figure 2. The N32, 64 and O32 C++ Compilation Processes 15

Tables

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

Preface

This publication documents the 7.3 release of the MIPSpro C++ compiler, which is invoked by the `CC`(1) command.

The MIPSpro C++ compiler runs on IRIX 6.2 and later operating systems.

Related Publications

The following documents contain additional information that may be helpful:

- *dbx User's Guide*
- *MIPSpro Compiling and Performance Tuning Guide*
- *Developer Magic: Debugger User's Guide*
- *MIPSpro N32 ABI Handbook*
- *MIPSpro 64-Bit Porting and Transition Guide*

Obtaining Publications

The *User Publications Catalog* describes the availability and content of all Cray Research hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, call +1 651 683 5907. Silicon Graphics employees may send electronic mail to `orderdsk@sgi.com` (UNIX system users).

Customers who subscribe to the CRInform program can order software release packages electronically by using the `Order Cray Software` option.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:
	1 — User commands
	1B — User commands ported from BSD
	2 — System calls
	3 — Library routines, macros, and opdefs
	4 — Devices (special files)
	4P — Protocols
	5 — File formats
	7 — Miscellaneous topics
	7D — DWB-related information
	8 — Administrator commands
	Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

The default shell in the UNICOS and UNICOS/mk operating systems, referred to in Cray Research documentation as the *standard shell*, is a version of the Korn shell that conforms to the following standards:

- Institute of Electrical and Electronics Engineers (IEEE) Portable Operating System Interface (POSIX) Standard 1003.2–1992
- X/Open Portability Guide, Issue 4 (XPG4)

The UNICOS and UNICOS/mk operating systems also support the optional use of the C shell.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send electronic mail to the following address:

  `techpubs@sgi.com`

- Send a facsimile to the attention of "Technical Publications" at fax number +1 650 932 0801.
- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

  For Silicon Graphics IRIX based operating systems: 1 800 800 4SGI

  For UNICOS or UNICOS/mk based operating systems or CRAY Origin2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

  Technical Publications
  Silicon Graphics, Inc.
  1600 Amphitheatre Pkwy.
  Mountain View, California 94043–1351

We value your comments and will respond to them promptly.

Understanding the Silicon Graphics C++ Environment [1]

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

- Section 1.1, page 1, discusses the three C++ compilers Silicon Graphics provides for IRIX 6.x systems.
- Section 1.2, page 2, describes the application binary interfaces and instruction set architectures supported by the MIPSpro compilers.
- Section 1.3, page 4, describes the differences between the N32, 64, and the O32 compilers.
- Section 1.4, page 5, describes the features and ABI changes that were available starting with the 6.2 and 7.0 versions of the MIPSpro C++ compilers.
- Section 1.5, page 12, discusses the C++ libraries in the Silicon Graphics C++ environment.
- Section 1.6, page 12, discusses the Silicon Graphics C++ debugging environment.

1.1 Silicon Graphics C++ Environment

The Silicon Graphics 7.3 C++ environment provides three C++ compilers for IRIX 6.x systems. As shown in Figure 1, page 2, there are two MIPSpro C++ compilers, a 32-bit version and a 64-bit version. They are known as the N32 and 64 compilers because of the *application binary interface*s (ABIs) they generate. These compilers accept a dialect of C++ that closely resembles the ANSI/ISO draft C++ standard and are strongly recommended by Silicon Graphics.

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. The C++ compiler based on Cfront is still available but is no longer supported. See the C++ release notes for the latest information about the status of these compilers.

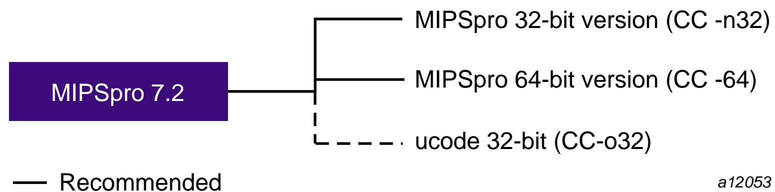

Figure 1. Silicon Graphics C++ Environment

The following commands invoke the three compilers:

`CC -n32`	32-bit native MIPSpro compiler. Generates N32 ABI objects.
`CC -64`	64-bit native MIPSpro compiler. Generates 64 ABI objects.
`CC -o32` or `CC -32`	32-bit native ucode compiler. Generates O32 ABI objects.

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

1.2 Understanding ABIs and ISAs

An application binary interface (ABI) 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 7.3 C++ are N32, 64, and O32. See Table 1, page 3, for a summary of the ABIs, their features, and their relationships to the MIPS ISAs.

Table 1. Features of the Application Binary Interfaces

	N32 (`-n32`)	64 (`-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 2, page 3, shows the MIPS ISAs and the MIPS processors for which they were designed.

Table 2. ISAs and Targeted MIPS Processors

ISA	Target Processor
MIPS IV	R5000, R8000, R10000, R12000
MIPS III	R4000 (Rev 2.2 and later), R4400, R46000
MIPS II	R4000 (Rev. 2.1 and earlier)
MIPS I	R2000, R3000

Table 1, page 3, and Table 2, page 3 are intended to give an overview of ABIs and MIPS ISAs. They do not show such details as the default ISAs shown in Table 1, page 3, can be changed with the environmental variable `SGI_ABI`. Also, these tables do not indicate some facts such as an R10000-based, IRIX 6.3 O2 system does not support 64 MIPS IV, while an R10000-based IRIX 6.4, OCTANE system does support it. The release notes for your system, the *MIPSpro Compiling and Performance Tuning Guide*, and the `cc`(1), `CC`(1), and `ABI`(5) man pages, provide more information about these details.

1.3 N32, 64, and O32 Compilation

The differences between the N32, 64 and the O32 compilers include the following:

- The code generated for N32 and 64 is much more highly optimized than that generated for O32. However, it may take longer to compile code for N32 and 64 due to the increased optimization performed.
- The warning options used by the `-woff` option are different.

The following are the default compilation modes:

- `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, R10000, and R12000 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, R10000, and R12000 systems.

The general relationship between ABIs, ISAs, and the CPUs that can run them is shown in Table 3, page 4. Again, because of system variations, there are some exceptions to the combinations shown. Consult your system's man pages and release notes for more information.

Table 3. ISAs, ABIs, and Their Host CPUs

	`-n32`	`-64`	`-o32`
-mips4	R12000, R10000, R8000, R5000	R12000, R10000, R8000	
-mips3	R10000, R8000, R5000, R4600, R4400, R4000 (>=Rev. 2.2)		

-n32	-64	-o32
-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.

See the following sources for additional information about O32, N32, and 64 compiling:

- 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 more information on N32 and 64 compilers.

1.4 Feature and ABI Changes in the MIPSpro C++ Compilers

Both the N32 and 64 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 the `#ifdef` construct if you want your code to be portable across all Silicon Graphics C++ compilers.

There is a builtin macro, `__EDG_ABI_COMPATIBILITY_VERSION`, which is undefined in the O32 compiler, but set to the integer value 229 in the 64 and N32 compilers. You can either use this macro to protect your use of the language features of the N32 and 64 compilers, or you can create your own `#ifdef` construct.

1.4.1 OpenMP API Multiprocessing Directives

MIPSpro C and C++ compilers support directives based on the OpenMP C/C++ Application Program Interface (API) standard. Programs that use these directives are portable and can be compiled by other compilers that support the OpenMP standard.

To enable recognition of the OpenMP directives, specify `-mp` on the `cc` or `CC` command line.

In addition to directives, the OpenMP C/C++ API describes several library functions and environment variables. Information on the library functions can be found on the `omp_lock`(3), `omp_nested`(3), and `omp_threads`(3) man pages. Information on the environment variables can be found on the `pe_environ`(5) man page.

See the *MIPSpro C and C++ Pragmas* manual for definitions and details about how to use the OpenMP `#pragma` directives.

1.4.2 Operators `new[ ]` and `delete[ ]`

The N32 and 64 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 do not 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 following 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.

1.4.3 Built-in `bool` Type

There is a built-in `bool` type in the 64 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, which 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 64 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.

1.4.4 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 64 compile and will not interfere with the built-in `wchar_t`. When the compiler supports `wchar_t`, it also defines a macro called `_WCHAR_T`; this allows you to write code that is portable across the O32 and the two MIPSpro compilers.

For instance, you can now read and write `wchar_t` types directly. They will not be read and written as `long` types, but will actually be read and written as multi-byte characters during 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, as in the following example:

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

wchar_t w;
foo(w);    // OK in -o32, 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 indicate that foo(long) has already been declared. This happens because in O32, wchar_t is a synonym for long.

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

1.4.5 Exception Handling

Exception handling is on by default in the MIPSpro 7.x compilers in the N32 and 64 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: 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 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.

1.4.6 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 want 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 2: Run-time Type Identification (RTTI)

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

You may want 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.

1.4.7 Other ABI Changes

There are three other ABI changes:

- 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 64 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 execute 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 64) 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.

1.5 C++ Libraries

By default, all C++ programs link with the complete C++ standard library which contains all of the `iostream` library functions, as well as the C++ storage allocation operators `::new` and `::delete`.

As of the 7.3 release, there are two complex libraries and two I/O libraries. The old complex library is used with `-o32`, which requires explicit linking using the `-lcomplex` option; the new complex library is used for `-n32` and `-64`, which is linked by default. The old I/O library is used for `-o32`, `-n32`, and `-64`; the new I/O library is used only for `-n32` and `-64` when the `-LANG:std` option is specified. You must continue to link with the math library by specifying the `-lm` option.

Silicon Graphics also provides the complex arithmetic library. To use this package with the `-o32`, you must explicitly link with this library. For example,

```
CC -o32 complexapp.c++ -lcomplex
```

For more information on the `complex` and `iostream` libraries, see the C++ standard.

1.6 Debugging

You can debug your C++ programs with `dbx` (a source-level debugger for C, C++, Fortran, and assembly language) or with the MIPSpro 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*.

Compiling, Linking, and Running C++ Programs [2]

This chapter contains two major sections:

- Section 2.1, page 13, describes the Silicon Graphics C++ compilation process.
- Section 2.2, page 17, briefly summarizes the capabilities of the tools that provide symbol and other information on object files.

2.1 Compiling and Linking

This section discusses C++ compiling and linking for the N32, 64, and O32 compilers.

The `CC`(1) command (see the `cc`(1) man page) invokes the C++ compiler. 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 compiler options, which give instructions to the processing phases. Options can appear anywhere in the command line.
filename	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 compiler. See the `cc`(1) man page 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:...`) (See also the `ipa`(5) man page.)

- Loop nest optimization (-LNO:...)(See also the lno(5) man page.)
- Floating point and miscellaneous optimizations (-OPT:...) (See also the opt(5) man page.)
- Precompiled headers

2.1.1 Compilation

The two compilation processes in Figure 2, page 15, show what transformations a C++ source file, foo.C, undergoes with the N32, 64, and O32 compilers. The MIPSpro compilation process is on the left, invoked by specifying either -n32 or -64 mode:

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

On the right is the O32 compilation process, invoked with -o32:

```
CC -o32 -o foo foo.C
```

The following steps further describe the compilation stages in Figure 2, page 15:

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++ compiler.
3. The complete source is processed using 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. Optimized object code (foo.o) is then generated.

   Note: To stop the compilation at this stage, use the following command line:

   CC *mode* -c foo.C

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

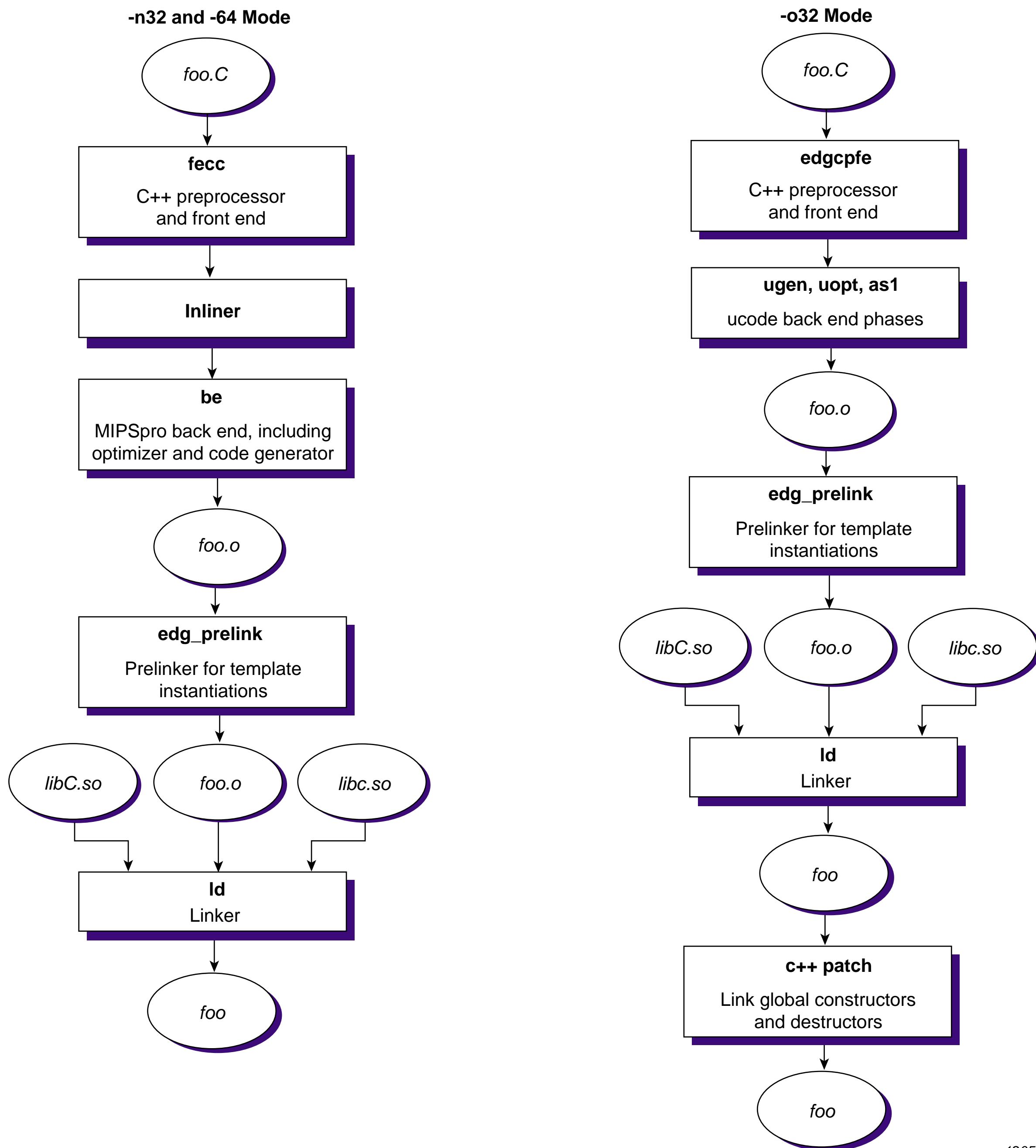

Figure 2. The N32, 64 and O32 C++ Compilation Processes

5. The compiler processes the `.ii` files associated with the objects that will be linked together. Then sources are recompiled to force template instantiation.
6. The object files are sent to the linker, `ld` (see the `ld`(1) man page) 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.

2.1.2 Sample Command Lines

The following are some typical C++ compiler command lines:

- 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 ...
```

2.1.3 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 the following:

1. Compile each program module separately with the appropriate compiler.
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 Section 1.5, page 12.

2.2 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 /usr/lib/c++ directory. For example:

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

libmangle.a
: The /usr/lib/c++/libmangle.a library provides a demangle(char *) function 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 must declare the following in your program, and link with the library using the -lmangle option:

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

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 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) man page 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) man page 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) man page for more information.

For more complete information on the object file tools, see the *MIPSpro Compiling and Performance Tuning Guide*.

C++ Dialect Support [3]

This chapter describes the C++ language implemented by the new 32–bit (`-n32`)and 64–bit (`-64`) MIPSpro C++ compilers. The old 32–bit (`-o32`) compiler accepts an older version of the C++ language, which is not discussed in this chapter.

This chapter contains the following major sections:

- Section 3.1, page 19, contains background information on the MIPSpro C++ compiler.
- Section 3.2, page 20, contains a list of MIPSpro C++ compiler features that are not in the *Annotated C++ Reference Manual* (ARM), but are in the C++ International Standard.
- Section 3.3, page 22, contains a list of features not supported by the MIPSpro C++ compilers that are not in the ARM, but are in the C++ International Standard.
- Section 3.4, page 23, contains a list of the anachronisms that are supported in the MIPSpro compilers when the `-anach` option is enabled.
- Section 3.5, page 24, contains a list of the extensions that are accepted by the MIPSpro compilers by default.
- Section 3.6, page 25, contains a list of extensions accepted by the MIPSpro compilers in Cfront-compatibility mode.
- Section 3.7, page 29, contains a list of constructs that Cfront supports but the MIPSpro compilers reject.

3.1 About the Compiler

The C++ compiler 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 C++ International Standard.

The `-anach` command line option enables anachronisms from older C++ implementations. By default, anachronisms are disabled. See Section 3.4, page 23, for details.

By default, the compiler 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 Section 3.5, page 24, for details.

The compiler 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 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 compiler does not support Cfront compatibility. See Section 3.6, page 25, and Section 3.7, page 29, for details.

3.2 New Language Features

The following features, not in the ARM but in the C++ International Standard, 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 structs, 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.

3.3 Non-implemented Language Features

The following features which are in the C++ International Standard but are not in the ARM, are not accepted by the MIPSpro C++ compilers:

- `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 a member of another class if the classes are unrelated.
- 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, however, the C++ International Standard requires that `f()` be evaluated.
- Class name injection 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.

3.4 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
}
```

3.5 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.

3.6 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 C++ International Standard.
- 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 in the following example:

```
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
```

3.7 Cfront Compatibility Restrictions

Even when you specify the `-cfront` option, the N32, 64, and O32 C++ compilers are not completely backwards-compatible with Cfront. The N32, 64, 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 uses the standard UNIX `cpp` and 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. The `CC` command does not strip the type signature when you are building the mangled name. If you try to do so, the following warning is issued:

  ```
  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.

Using Templates [4]

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:

- Section 4.1, page 31, describes how to perform template instantiation in the Silicon Graphics C++ environment.
- Section 4.2, page 44, 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.
- Section 4.3, page 47, 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 7.3 Release Notes have information about the particular version of the STL being shipped with the 7.3 release.

4.1 Template Instantiation

The instantiation of a class template is 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 for the following reasons:

- 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.

4.1.1 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.

The `CC`(1) command 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.

4.1.1.1 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 encounters 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 Section 4.1.2, page 35.
- You can make sure that the files that define template entities also have the definitions of all the available types, and add code or `#pragma` directives in those files to request instantiation of the entities they contain.

4.1.1.2 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 (for example, `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 a file is compiled, the compiler 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.

4.1.1.3 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 the following conditions 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, such as the following:

```
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 #pragma directives 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.

4.1.2 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 the `xyz.h` file.
- 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.

4.1.3 Explicit Instantiation

The `CC` command instantiates all templates at compile time if you specify the `-ptused` option. The compiler produces larger object files because it stores duplicate instantiations in the object files. Duplicate copies may not be removed by the linker and may exist in the final executables.

The `CC` template instantiation mechanism also correctly handles static data members when you use the `-ptused` option. Static data members that must 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 the make(1) command in conjunction with the -MDupdate option. See the CC(1) reference page and Section 4.1.6, page 42, for more information.)
- If you plan on specializing a template function instantiation, you may have to set the #pragma do_not_instantiate directive 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 #pragma directives and command-line options.

4.1.3.1 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 in the following list. You use one option from each category: options from the same category are not used together. (For example, you cannot specify -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 following command line options control instantiation behavior of the compiler:

-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.
`-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 specify `-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 compiler 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 compiler to not invoke the template prelinker at link time.

Option	Description
	This is the default mode if `-ptused` or `-ptall` is specified.
`-auto_include`	Instructs the compiler to implicitly include template definition files if such definitions are needed. (See Section 4.1.2, page 35.)
`-no_auto_include`	Disables implicit inclusion of template implementation files. (See Section 4.1.2, page 35.)
`-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.

4.1.3.2 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 how they may be used.

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

Options	Description
`-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).
`-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, it is recommended 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 template 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 specify `-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.

4.1.3.3 `#pragma` Directives for Template Instantiation

You can use `#pragma` directives to control the instantiation of individual or sets of template entities. There are three instantiation `#pragma` directives:

`#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 `#pragma` directives 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 directive 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 #pragma do_not_instantiate directive as in the following example:

```
#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 #pragma instantiate directive 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 following example:

```
#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

4.1.4 Specialization

The `CC` command 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 Section 4.1.3.3, page 40, 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.

4.1.5 Building Shared Libraries and Archives

When you build a shared library or archive, you should instantiate any template instances that could be needed. You can have the prelinker automatically instantiate all needed templates in either of the following two ways:

- Build a shared library by using a command such as the following:

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

- Build an archive by using a command such as the following:

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

4.1.6 Limitations on Template Instantiation

The following subsections discuss the limitations on template instantiation in the Silicon Graphics C++ environment.

4.1.6.1 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 may not be selected, although a specialization should take precedence over a generated function.

The following conditions must 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.

4.1.6.2 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 dependence file (usually called `Makedepend`). The compiler lists dependences for all

applicable `#include` files, including template implementation files that are implicitly included.

2. Make sure that your `Makefile` includes this dependence file. See the `CC`(1) and `make`(1) man pages for more information on how to include files within a `Makefile`.

4.1.6.3 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 Section 4.1.5, page 42.)
- 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.

4.2 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.

4.2.1 Mapping Template Options From Cfront to CC

The following list contains Cfront template-related options, their meanings, and the equivalent `CC` options:

`-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 searches 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 the following 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 Section 4.2.2, page 46, and Section 4.2.3, page 47, for more 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 `#pragma` directives described in Section 4.1.3.3, page 40.

`-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.

4.2.2 Using Object Files From Cfront's Repository

If you are familiar with 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 want 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` compiler will automatically run the prelinker on the set of object files being linked into the shared libraries. No further action is necessary.
- If an archive is being built, the prelinker needs to be run explicitly on the object files, before invoking `ar`. See Section 4.1.5, page 42, for information on this action.

4.2.3 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.

4.3 Template Language Support

The language support for templates in the Silicon Graphics C++ environment is more extensive than for Cfront. The following are some of the additional template language constructs supported by the Silicon Graphics C++ environment:

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

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

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

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

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

```
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.

The Auto-Parallelizing Option (APO) [5]

Note: APO is licensed and sold separately from the MIPSpro C/C++ compilers. APO features in your code are ignored unless you are licensed for this product. For sales and licensing information, contact your sales representative.

The Auto-Parallelizing Option (APO) enables the MIPSpro C/C++ compilers to optimize parallel codes and enhances performance on multiprocessor systems. APO is controlled with command line options and source directives.

APO is integrated into the compiler; it is not a source-to-source preprocessor. Although run-time performance suffers slightly on single-processor systems, parallelized programs can be created and debugged with APO enabled.

Parallelization is the process of analyzing sequential programs for parallelism and restructuring them to run efficiently on multiprocessor systems. The goal is to minimize the overall computation time by distributing the computational workload among the available processors. Parallelization can be automatic or manual.

During *automatic parallelization*, the compiler analyzes and restructures the program with little or no intervention by you. With APO, the compiler automatically generates code that splits the processing of loops among multiple processors. An alternative is *manual parallelization*, in which you perform the parallelization using compiler directives and other programming techniques.

APO integrates automatic parallelization with other compiler optimizations, such as interprocedural analysis (IPA), optimizations for single processors, and loop nest optimization (LNO). In addition, run-time and compile-time performance is improved.

For details on using APO command line options and source directives, see the *MIPSpro C and C++ Pragmas* manual.

Index

32–bit ABI (ucode), 2
64
  vs. 32–bit, 4

A

ABI
  64, 1
  additional information, 5
  Cfront compatibility, 29
  commands, 2
  compilation process, 14
  definition, 2
  dialect support, 19
  differences, 4
  features, 3
  host CPUs, 4
  N32, 1
  N32 APO, 51
  N64 APO, 51
  new features, 5, 11
  O32, 1
allocator(), 9
Anachronisms, 19
  accepted, 23
  base class, 23
and, 22
-ansiE, 24
-ansiW, 24
APO, 51
  licensing, 51
Application binary interface
  See "ABI", 1
Application Program Interface, 5
Archives
  building, 47
ARM
  non-implemented features, 22
Auto-Parallelizing Option
  See "APO", 51
Automatic instantiation, 32
  suppressing, 38
Automatic parallelization
  definition, 51

B

Bit fields, 28
bitand, 22
Block
  catch, 9
  throw, 9
  try, 9
bool, 7, 21
Borland compilers, 31

C

C
  compile/link with C++, 16
C++
  command lines, 16
  environment, 1
  IRIX 6.x systems, 1
c++filt, 17
c++patch, 16
catch, 9, 23
CC
  syntax, 13
Cfront
  compatibility mode, 20
  compatibility restrictions, 29
  template transition, 44
Cfront compiler, 1

-cfront, 25
Command lines
examples, 16
Commands, 2
template instantiation, 36
Compatibility restrictions, Cfront, 29
Compilation, 14
process (figure), 16
to stop, 14
Compilation modes, 4
Compiler
Cfront, 2
ucode, 1
Compiling, 13
Complex arithmetic library, 12
complex libraries, 12
const volatile, 21, 22
const_cast, 21
Constructors, 16
trivial, 22
Conversion operators, 25
ctor(), 10
cv-qualifiers, 22

D

dbx, 12
Debugger
dbx, 12
WorkShop, 12
delete, 21, 22
delete[], 6
Demangling, 17
derived, 10
Destructors, 16, 22
derived class, 27
Developer Magic, 12
Dialect support, 19
Digraphs, 22
Directives
arguments, 40
OpenMP, 5
#pragma, 40
template class name, 41
#pragma can_instantiate, 40
#pragma do_not_instantiate, 36, 40
#pragma instantiate, 40
Drivers, 13
dwarfdump, 18
Dynamic shared object (DSO), 39
dynamic-cast, 10
dynamic_cast, 21

E

__EDG_ABI_COMPATIBILITY_VERSION, 5
eExtensions, 20
elfdump, 18
Errors, 20
O32, 7
Examples
anachronism, 24
c++ filt, 17
demangling, 17
linking with Fortran, 17
#pragma directives, 41
template language support, 48
typical command lines, 16
Exception handling, 5, 8
example, 9
explicit, 21
Explicit instantiation, 35
Extensions
Cfront, 25
Cfront mode, 25
default mode, 24

F

Features
anachronisms, 23
Cfront-compatibility extensions, 25

extensions, 24
new, 5, 20
non-implemented, 22
for-init-statement, 22
Fortran
compile/link with C++, 16
friend, 24, 26
Function
allocator(), 9
Functions
non-implemented, 22
operator++(), 23
operator–(), 23
overloading, 23

G

Global constructors, 16
Global destructors, 16

I

Implicit inclusion, 35
Inclusion, implicit, 35
Instantiation, 31
automatic method of, 33
automatic, details of, 33
requirements, 32
suppressing, 38
Instantiation, command-line options, 36
Instruction Set Architecture
See "ISA", 2
Instruction sets, 4
Interprocedural analysis (IPA), 14
iostream library, 12
IPA
automatic parallelization, 51
IRIX environment, 5
ISA
definition, 3

K

Keywords
bool, 7
typename, 21
unrecognized, 22

L

Languages
linking with other, 16
ld, 16
libc.so, 16
libmangle.a, 17
Libraries, 17
complex, 12
iostream, 12
libc.so, 16
new, 12
standard, 12
Library
building, 47
Link editor, 16
link libraries, 17
Link-time changes, 43
Linkage specification, 22
Linker, 16
Linking, 13
Cfront differences, 31
with other languages, 16
LNO
automatic parallelization, 51
Loader, 16
Loop nest optimization (LNO), 14

M

Makefile, 44
Mangled names, 11
Manual parallelization, 51

Mapping options
  from Cfront to CC, 44
Multi-language programs, 16
mutable, 21

N

N32
  See "ABI", 1
  vs. 64-bit, 4
Name mangling, 11
  differences, 30
Namespaces, 21
Nested classes, 21
new, 21, 22
New-style casts, 21
new[], 6
nm, 17

O

O32
  errors, 7
  See "ABI", 1
Object files, 13
  linking, 17
  tools, 17
    additional information, 19
    c++filt, 17
    dwarfdump, 18
    elfdump, 18
    libmangle.a, 17
    nm, 17
    size, 17
    stdump, 18
Object layout, 11
omp_lock, 6
omp_nested, 6
omp_threads, 6
OpemMP
  multiprocessing directives, 5
Operators
  ?, 26
  delete[], 6
  new[], 6
Optimization, 4, 13
  APO, 51
overload, 23

P

Parallelization
  automatic, 51
  definition, 51
  manual, 51
Pascal
  compile/link with C++, 16
pe_environ, 6
Point of definition (POD) constructs, 22
#pragma can_instantiate, 40
#pragma do_not_instantiate, 40
#pragma instantiate, 40
Prelink file, 14
Prelinker, 33
  recompiling files, 44
Preprocessor, 14
Processors
  MIPS, 3

Q

Qualified names, 20

R

reinterpret_cast, 21, 22
Renamed object file, 44
Repositories
  multiple, 47
Repository

object files, 46
RTTI
example, 10
new features, 21
Run-time type identification
See "RTTI", 10

S

Shared libraries, building, 42
size, 17
Source file, suffix, 14
Specialization, 42
Standard Template Library
See "STL", 31
Standards
C++ Internation Standard, 19
The Annotated C++ Reference Manual, 19
static_cast, 21
stdump, 18
suffixes, file, 14

T

Templates
<>, 22
archives, 42
unselected specializations, 43
automatic instantiation, 32
automatic instantiation method, 33
class, 22
command-line instantiation, 36
comparison, 31
explicit instantiation, 21, 35
implicit inclusion, 35
instantiation, 31
building library, 39
limitations, 43
specialization, 42
instantiation examples, 38
instantiation requirements, 32
language support, 48
link-time changes, 43
mapping Cfront to CC options, 44
member, 22
multiple repositories, 47
name binding, 23
new features, 21
object files in Cfront repository, 46
#pragma directives, 40
pre-instantiated, 39
shared library, 42
specialization, 32
transitioning from Cfront, 44
this, 25
throw, 9
Translators, 13
try, 9, 23
Type
bool, 7
type_info, 11
wchar_t, 7
type_info, 11
typedef, 29
typeid, 11, 21
typename, 21
Types
enum, 22

U

ucode compiler, 1

V

Virtual function tables, 11

W

Warnings, 20

wchar_t, 7, 21
_WCHAR_T, 7
WorkShop, 12