MIPSpro™ C++ Programmer's Guide

007–0704–150

CONTRIBUTORS

Rewritten in 2002 by Jean Wilson with engineering support from John Wilkinson and editing support from Susan Wilkening.

Cover design by Sarah Bolles, Sarah Bolles Design, and Dany Galgani, SGI Technical Publications.

New Features in this Guide

This guide has been rewritten to include information about the C++ command line, and to add further information about C++ pragmas.

Diagnostic information has also been added in Chapter 2, "Compiling, Linking, and Running Programs", page 9.

Record of Revision

Version	Description
7.3	April 1999 This revision supports the MIPSpro 7.3 release.
140	September 2002 This revision supports the MIPSpro 7.4 release which runs on the IRIX operating system, version 6.5 and later.
150	June 2003 This revision supports the MIPSpro 7.4.1 release which runs on the IRIX operating system, version 6.5 and later.

Contents

Figures

Tables

About This Guide

This publication documents the 7.4 release of the MIPSpro C++ compiler, which is invoked by the `CC`(1) command. This document describes the C++ compiling environment and the libraries used with C++.

The 7.4 version of the MIPSpro C++ compiler runs on IRIX 6.5 and later versions of the operating system.

Related Publications

The following documents contain information that may be helpful in porting code to the newer SGI compilers:

- *MIPS O32 Compiling and Performance Tuning Guide*
- *MIPSpro N32/64 Compiling and Performance Tuning Guide*
- *MIPSpro N32 ABI Handbook*
- *MIPSpro 64-Bit Porting and Transition Guide*

The following documents contain information about SGI's implementation of C and C++:

- *C Language Reference Manual*
- *MIPSpro C and C++ Pragmas*

Several performance evaluation and debugging tools are available to help you optimize and evaluate your code. See the *ProDev WorkShop: Overview* for a description of the different tools that are available.

See the *Guides to SGI Compilers and Compiling Tools* for an overview of all SGI compilers, compiler documentation, optimization tools, porting tools, and performance tools.

In addition to the above SGI documentation, several third party documents contain additional information which may be helpful. These books can be ordered from any book vendor:

- Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, special edition, 2000. ISBN 0201700735.
- Josuttis, Nicolai. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Publishing Company, 1999. ISBN 0201379260.

The C++ Standard, ISO/IEC 14882, *Information Technology — Programming Languages — C++* is available from the American National Standards Institute at `http://www.ansi.org`.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- See the SGI Technical Publications Library at `http://docs.sgi.com`. Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.
- If it is installed on your SGI system, you can use InfoSearch, an online tool that provides a more limited set of online books, release notes, and man pages. With an IRIX system, select **Help** from the Toolchest, and then select **InfoSearch**. Or you can type `infosearch` on a command line.
- You can also view release notes by typing either `grelnotes` or `relnotes` on a command line.
- You can also view man pages by typing `man` *title* on a command line.

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.
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.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in any of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Use the Feedback option on the Technical Publications Library Web page:

  `http://docs.sgi.com`

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system.

- Send mail to the following address:

  Technical Publications
  SGI
  1600 Amphitheatre Parkway, M/S 535
  Mountain View, California 94043–1351

- Send a fax to the attention of "Technical Publications" at +1 650 932 0801.

SGI values your comments and will respond to them promptly.

Overview of the SGI C++ Environment

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

- "The Compiler Programming Environment", discusses the different elements included in the compiler environment.
- "The SGI C++ Compilers", page 2, discusses the compilers that SGI provides for IRIX 6.*x* systems.
- "OpenMP API Multiprocessing Directives", page 6, discusses the OpenMP features available with the MIPSpro C++ compilers.
- "C++ Libraries", page 7, discusses the C++ libraries in the SGI C++ environment.
- "Debugging", page 8, discusses the SGI C++ debugging environment.

The Compiler Programming Environment

Note: The MIPSpro C++ compiler is based on the Edison Design Group (EDG) C++ Front End, Version 2.45.

The compiling environment allows you to develop, debug, and run C++ codes on your computer system. It includes the following products:

- A compiler, including the C++ front end, the language-independent optimizing back end, the linker, and other compiling elements.
- The libraries. SGI supports the C++ Standard Library as well as the C library, including the interface for C++ embedded in such header files as `cstdlib`. Information on the individual C library functions can be found in the online man pages for each function.
- The performance tools contained in SpeedShop and in the ProDev WorkShop suite. For more information on these products, see the *SpeedShop User's Guide* or the *ProDev WorkShop: Overview*.
- An archiving tool. An *archive library* is a file that contains one or more routines in object file format (*file*`.o`). When a program calls an object file that is not explicitly

included in the program, the linker, `ld`(1), looks for that object file in an archive library. The scheduler then loads only that object file, not the whole library, and loads it with the calling program.

The archiver creates and maintains archive libraries. It allows you to copy new objects into the library, replace existing objects in the library, move objects within the library, and copy individual objects from the library into individual object files. For more information on the archive library, see the `ar`(1) man page.

- Object file tools, which allow you to disassemble object files into machine instructions, print information about archive files, and perform other tasks. For more information on these tools, see the following man pages: `dis`(1), `elfdump`(1), `file`(1), `nm`(1), `size`(1), and `strip`(1).
- Online documentation utilities. The `man`(1) command allows you to retrieve online man pages. Prose reference text, such as this manual, can be retrieved through the Web browser supported at your site. See `http://www.techpubs.sgi.com` for all online documentation.
- Modules. The compiler can be installed with the modules utility. This utility allows you to access different versions of the compiler and runtime environment. For more information on using the modules utility, see the `modules`(1) man page or enter the following command:

  ```
  % relnotes modules
  ```

- The message system. This system lets you obtain more comprehensive explanations of messages generated by the compiler and tools in the compiling environment. When a message condition occurs, both a message number and a verbal summary of the problem is generated. If you need more information on the error condition described in the summary, you can enter the `explain`(1) command to retrieve a more detailed description.
- Environment variables. For more information, see the `pe_environ`(5) man page, which describes many environment variables that can be used when compiling C++ programs.

The SGI C++ Compilers

The SGI C++ environment provides three C++ compilers for IRIX 6.*x* systems. As shown in Figure 1-1, page 3, 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 implement the C++ dialect as documented in the the ANSI/ISO C++ standard (hereafter referred to as "The Standard").

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 now unavailable and is no longer supported. See the C++ release notes for the latest information about the status of these compilers.

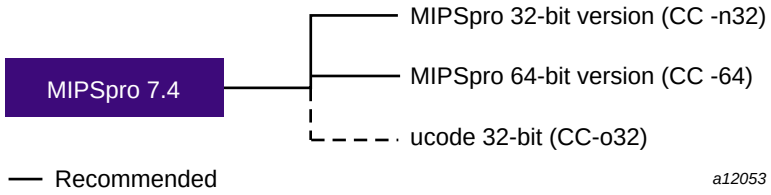

Figure 1-1 SGI C++ Environment

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 the compilers and ABIs, see the `CC`(1) and `ABI`(5) man pages, or the *MIPSpro N32/64 Compiling and Performance and Tuning Guide*.

The focus of discussion in this document centers on the 32–bit and the 64–bit native MIPSpro compilers. For details about the 32–bit ucode compiler, see the *MIPS O32 Compiling and Performance Tuning Guide*.

Understanding ABIs and ISAs

An ABI defines a system interface for executing compiled programs. Among the important features that 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 C++ are N32, 64, and O32. This document discuss the N32 and 64 ABIs. See Table 1-1, page 4, for a summary of these ABIs, their features, and their relationships to the MIPS ISAs. The O32 ABI is not discussed in detail in this document.

Table 1-1 Features of the Application Binary Interfaces

	N32 (`-n32`)	64 (`-64`)
Default ISA	MIPS III	MIPS IV
Alternate ISA (Option)	MIPS IV (`-mips4`)	MIPS III (`-mips3`)
Floating Point Registers	32	32
Register Size	64 bits	64 bits
`int`	32 bits	32 bits
`long int`	32 bits	64 bits
`char*`	32 bits	64 bits

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

Table 1-2 ISAs and Targeted MIPS Processors

ISA	Target Processor
MIPS IV	R5000, R8000, R10000, R12000, R14000
MIPS III	R4000 (Rev 2.2 and later), R4400, R46000
MIPS II	R4000 (Rev. 2.1 and earlier)
MIPS I	R2000, R3000

Table 1-1 and Table 1-2 are intended to give an overview of ABIs and MIPS ISAs. They do not show details (such as an R10000-based, IRIX 6.3 Silicon Graphics O2 system does not support 64 MIPS IV, but an R10000-based, IRIX 6.4 Silicon Graphics Octane system does support it).

For more information about ABIs and ISAs, see the release notes for your system, the *MIPSpro N32/64 Compiling and Performance and Tuning Guide*, the *MIPSpro N32 ABI Handbook* and the `cc`(1) and `ABI`(5) man pages.

N32 and 64 Compilation

The following are the default compilation modes for the different supported ABIs:

- `CC -64` is `-mips4`
- `CC -n32` is `-mips3`

SGI 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 the N32 and 64 ABIs, ISAs, and the CPUs that can run them is shown in Table 1-3, page 6. 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 1-3 ISAs, ABIs, and their Host CPUs

	-n32	-64
-mips4	R14000, R12000, R10000, R8000, R5000	R12000, R10000, R8000
-mips3	R10000, R8000, R5000, R4600, R4400, R4000 (>=Rev. 2.2)	

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:

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

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.

C++ Libraries

In addtion to the compiler itself, the MIPSpro C++ product includes the C++ Standard Library, as described in Clauses 17–27 of the Standard. It consists of the following components:

- The runtime library, packaged as the dynamic shared object `libC.so`, provides support for language features such as storage allocation and deallocation, exception handling, and runtime type identification.
- The Standard Template Library (STL) comprises the headers that support the standard containers, iterators, and algorithms described in Clauses 23–25 of the Standard.
- The `iostream`, `string`, and `locale` libraries provide components for manipulation of character and wide character strings and input/output, including interationalization support. These components are made into templates and are packaged primarily as standard headers, but there is a runtime component packaged as the dynamic shared object `libCio.so`.
- The `valarray` and `complex` components provide support for potentially parallelizable vector arithmetic and for complex numbers, respectively.

MIPSpro C++ also provides the standard C headers, which are the interface between C++ and the C library. For backward compatibility, MIPSpro C++ continues to provide the old `iostream` and `complex` libraries, but their use is deprecated and only limited support is provided for them.

SGI also provides the C99 library of extended functions for version 7.4 (and later versions) of the MIPSpro compilers. The C99 library includes the following features:

- `_bool`, `_complex`, and `_imaginary` data types
- support for the `inline` keyword
- variable declaration in `for` loops
- support for compound literals
- support for the `_func_` predefined identifier
- use of designated initializers
- support for the `restrict` keyword

See the release notes provided with your compiler for a detailed list of C99 features.

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

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/tech/stl/`.

Debugging

You can debug your C++ programs by using `dbx` (a source-level debugger for C, C++, Fortran, and assembly language) or the MIPSpro WorkShop Debugger (a graphical, interactive, source-level debugging tool). For more information on the different available debugging tools, see the *ProDev WorkShop: Overview*.

Chapter 2

Compiling, Linking, and Running Programs

This chapter discusses C++ compiling and linking for the MIPSpro compilers. It contains the following sections:

- "The C++ Command Line", page 9, discusses the syntax of the C++ command and the options used with that command.
- "Compiling and Linking", page 21, describes the SGI C++ compilation process.
- "Object File Tools", page 29, briefly summarizes the capabilities of the tools that provide symbol and other information on object files.

The C++ Command Line

This section provides an overview of the `CC`(1) command. For complete details about each option, see the `CC` man page.

The `CC` command invokes the compiler. The following syntax box shows the complete `CC` command syntax (note that `cc` command syntax is not indicated here).

```
CC [-64|-n32] [-all] [-anach] [-ansiE] [-ansiW] [-apo] [-apokeep]
  [-apolist] [-ar] [-auto_include] [-bigp_off] [-bigp_on]
  [-brief_diagnostics] [-c] [-cfront] [-common] [-D name=def]
  [-DEBUG] [-diag_error numberlist] [-diag_remark numberlist]
  [-diag_suppress numberlist] [-diag_warning numberlist] [-dollar] [-E]
  [-fb file] [-fb_create path] [-fb_opt path] [-fbgen] [-fbuse file]
  [-FE:eliminate_duplicate_inline_copies]
  [-FE:template_in_elf_section] [-float] [-float_const][-fullwarn]
  [-G num] [-g[n]] [-gslim] [-help] [-I dir] [-ignore_suffix]
  [-INLINE] [-IPA] [-J #] [-KPIC] [-L directory] [-l library] [-LANG]
  [-LIST] [-LNO] [-M] [-MDupdate filename] [-mipsn] [-MP] [-mp]
  [-mplist] [-no_auto_include] [-noinline] [-non_shared]
  [-no_prelink] [-none] [-nostinc] [-o outfile] [-O[n]]
  [-Ofast [= ipxx]] [-OPT:] [-P] [-pch] [-pedantic] [-prelink]
  [-pta] [-ptall] [-ptnone] [-ptused] [-ptv] [-quiet_prelinker] [-r]
  [-rprocessor] [-S] [-shared] [-show] [-signed] [-TARG] [-TENV]
  [-trapuv] [-U name] [-use_command] [-use_readonly_const]
  [-use_readwrite_const] [-use_suffix] [-v] [-version]
  [-W c,arg1 [,arg2...]] [-w] [-w2] [-woff all] [-woff numberlist]
  [-x lang] [-Xcpluscomm] [-Y c,path] files
```

In some cases, more than one option can have an effect on a single compiler feature. The following list shows some of the compiler features and the options that affect them:

- Source preprocessing: `-D`*var*[=*def*][,*var*[=*def*]]..., `-E`, `-nocpp`, `-P`, `-U`*name*.
- Setting the compilation environment: `-n32`, `-64`, `-mips`*n*, `-r`*processor*, `-TARG:`, `-TENV:`.
- Optimization: `-apo`, `-LNO:`, `-OPT:`, `-O`*level*.

Note: In order to use the APO command line options, you must be licensed for the MIPSpro Auto-Parallelizing Option.

Various environment variable settings can affect your compilation. For more information on the environment variables, see the `pe_environ`(5) man page.

Some `CC`(1) command options, for example, `-LNO:...`, `-LIST:...`, `-MP:...` , `-OPT:...`, `-TARG:...`, and `-TENV:...` accept several suboptions and allow you to specify a setting for each suboption. To specify multiple suboptions, either use colons

to separate each suboption or specify multiple options on the command line. For example, the following command lines are equivalent:

```
% CC -LIST:notes=ON:options=OFF b.c
% CC -LIST:notes=ON -LIST:options=OFF b.c
```

Some arguments to suboptions of this type are specified with a setting that either enables or disables the feature. To enable a feature, specify the suboption either alone or with `=1`, `=ON`, or `=TRUE`. To disable a feature, specify the suboption with either `=0`, `=OFF`, or `=FALSE`. For example, the following command lines are equivalent:

```
% CC -LNO:auto_dist:blocking=OFF:oinvar=FALSE a.c
% CC -LNO:auto_dist=1:blocking=0:oinvar=OFF a.c
```

For brevity, this manual shows only the `ON` or `OFF` settings to suboptions, but the compiler also accepts 0, 1, `TRUE`, and `FALSE` as settings.

Command Line Options

The following list summarizes the options to the `CC` command. For complete details, see the `CC`(1) man page.

`-n32, -64`

Specifies the Application Binary Interface (ABI), either `-n32` or `-64`. Specifying `-n32` generates 32-bit objects. Specifying `-64` generates 64-bit objects.

`-all`

Specifies that the linker should pull in the entire archive into the shared object, not just the object files needed for the link.

`-anach`

Specifies that anachronistic C++ constructs are allowed.

`-ansiE`

Issues an error message on all code that is not standard-conforming.

`-ansiW`

Issues a warning message on all code that is not standard-conforming.

`-apo`, `-apokeep`, `-apolist`

Controls the Auto-Parallelizing Option (APO), which automatically converts sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so.

Note: These options are ignored unless you are licensed for the Auto-Parallelizing Option. For more information on this product, contact your sales representative.

`-ar`

Creates an archive instead of a shared object or executable.

`-auto_include`, `-no_auto_include`

Specifies that the compiler implicitly includes template definition files if such definitions are needed.

`-bigp_on`

Instructs that the compiler to enable the use of large pages within your program.

`-bigp_off`

Instructs that the compiler to disable the use of large pages within your program. This is the default for all optimization levels except `-Ofast`.

`-brief_diagnostics`

Issues one-line diagnostic messages.

`-c`

Disables the load step and writes the binary object file to *file*`.o`.

`-cfront`

Causes the compiler to accept constructs that were accepted by previous Cfront-based compilers.

`-common`

Relaxes the ANSI/ISO C Strict-Ref/Def-initialization model to the traditional IRIX Relaxed Ref/Def model.

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

Defines *name* to the macro preprocessor.

`-DEBUG:...`

Controls the compiler's attempts to detect various errors (at compile time or run time) and controls how the errors are reported. For more information on the debugging options, see the `debug_group`(5) man page.

`-diag_error` *numberlist*

Treats messages with the specified numbers as errors and does not generate an object file.

`-diag_remark` *numberlist*

Treats messages with the specified numbers as remarks unless they are nondiscretionary errors.

`-diag_suppress` *numberlist*

Equivalent to specifying `-woff`.

`-diag_warning` *numberlist*

Treats messages with the specified numbers as warnings unless they are nondiscretionary errors.

`-dollar`

Allows the dollar sign as a character in C identifiers.

`-E`

Run only the source preprocessor files, without considering suffixes, and writes the result to `stdout`.

`-fb`*file*

Specifies the feedback file to be used.

`-fb_create` *path*

Generates an instrumented executable program, which is suitable for producing one or more `.instr` files for subsequent feedback compilation.

`-fb_opt` *path*

Specifies the directory that contains the instrumentation output generated by compiling with `-fb_create` and then running your program with a training input set.

`-fbgen`

Generates an instrumented executable program.

`-fbuse` *file*

Specifies a `.Counts` file that is used to guide feedback compilation.

`-FE:eliminate_duplicate_inline_copies`

Eliminates duplicate copies of functions that are declared inline but for which an out-of-line copy must be generated.

`-FE:template_in_elf_section`

Eliminates duplicate template instantiation from an executable or DSO.

`-float`

Causes the compiler to use single-precision floating-point wherever float is specified, except in function arguments.

`-float_const`

Interprets floating point constants without precision suffixes as single-precision whenever doing so will not lose precision and the context is otherwise single-precision.

`-fullwarn`

Requests that the compiler generate comment-level messages. These messages are suppressed by default. This option can be useful during software development.

`-G`*num*

Specifies the maximum size, in bytes, of a data item that is to be accessed from the Global Pointer (GP). *num* must be a decimal number.

`-g`[*n*]

Generates debugging information and establishes a debugging level.

`-gslim`

Limits the amount of debugging information generated by the compiler for class definitions

`-help`

Lists all available options. The compiler is not invoked.

`-I`*dir*

Specifies a directory to be searched for `#include` files.

`-ignore_suffix`

Determines the language of the source file being compiled by the command used to invoke the compiler.

`-INLINE:…`

Specifies actions for the standalone inliner. For more information on the individual options in this group, see the `ipa`(5) man page.

`-IPA`[`:…`]

Controls the application of interprocedural analysis (IPA) and optimization. This includes inlining, common block array padding, constant propagation, dead function elimination, alias analysis, and other features. Specify `-IPA` with no arguments to invoke the interprocedural analysis phase with default options. For more information on the individual options in this group, see the `ipa`(5) man page.

`-J`[#]

Specifies the maximum number of concurrent compiles that the C++ prelinker is allowed to run at once.

`-KPIC`

Generates position-independent code (PIC), which is necessary for programs loaded with dynamic shared libraries. Enabled by default.

`-l`*library*

Searches the library named `lib`*library*`.a` or `lib`*library*`.so`. Libraries are searched in the order given on the command line.

`-L`*directory*

Changes the library search algorithm for the loader.

`-LANG:...`

Controls the language option group.

`-LIST:...`

Writes an assembler listing file to *file*`.l`.

`-LNO:`…

Specifies options and transformations performed on loop nests by the Loop Nest Optimizer. For details about these options, see the `lno`(5) man page.

`-M`[]

Runs only the preprocessor on the named files and writes `make` dependencies to standard output.

`-MDupdate`[*file*]

Updates makefile dependencies in *file*.

`-mips`*n*

Specifies the instruction set architecture (ISA).

`-mp`

Generates multiprocessing code for the files being compiled. This option causes the compiler to recognize all multiprocessing directives and enables all `-MP:...` options.

`-MP:…`

Specifies individual multiprocessing options that provide fine control over certain optimizations.

`-mplist`

Generates *file*`.w2f.c`.

`-noinline`

Suppresses expansion of inline functions.

`-non_shared`

Builds a nonshared object.

`-none`

Turns off the effects of `-all` for the remainder of the command line.

`-nostdinc`

Directs the system to skip the standard directory, `/usr/include`, when searching for `#include` files.

`-o`*outfile*

Writes the executable file to *outfile* rather than to `a.out`. By default, the executable output file is written to `a.out`.

`-O`*n*

Specifies the basic optimization level.

`-Ofast`[`=`*ipxx*]

Selects optimization that maximizes performance for a given SGI platform.

`-OPT:…`

Controls miscellaneous optimizations. These options override defaults based on the main optimization level. For details, see the `opt`(5) man page.

`-pch`

Uses the `-LANG:pch` option.

`-pedantic`

Warns that the `#ident` preprocessor directive is nonstandard.

`-prelnk`, `no_prelink`

Instructs the compiler to emit information in the object file and in an associated `.ii` file to help the prelinker determine which files are responsible for instantiating the various template entities referenced in a set of object files.

`-P`

Runs only the source preprocessor and puts the results for each source file in a corresponding *file*`.i`. The *file*`.i` that is generated does not contain # lines.

`-pta`, `ptall`

Instantiates template entities declared or referenced in the current compilation.

`-ptnone`,

No templates are instantiated.

`-ptused`,

Template entities used in this compilation will be instantiated.

`-ptv`,

Prints the name of the template entity and source file.

`-quiet_prelinker`,

Disables prelinker output in C++.

`-r`,

Creates Executable Linking Format (ELF) relocatable objects when specified with `-IPA` during compilation.

`-r`*processor*

Specifies the code scheduler.

`-S`

Generates an assembly file, *file*`.s`, rather than an object file (*file*`.o`). See the *MIPSpro Assembly Language Programmer's Guide* for a discussion of the assembly language file that can be created by using this option.

`-shared`

Creates a shared library instead of an executable program.

`-show`

Print the passes as they execute with their arguments and their input and output files.

`-signed`

Causes values of type `char` to be treated as if they had type `signed char`.

`-TARG:…`

Cross compiling is compiling a program on one system and executing it on another. To cross compile, you can either use the `-TARG:` command line options to control the target architecture and machine for which code is generated or you can set the `COMPILER_DEFAULTS_PATH` environment variable to specify the file that contains the default processor information needed to generate executable code for the target system.

`-TENV:…`

Specifies the target environment option group. The *target environment* is the system upon which the executable code will be run. These options control the target environment assumed and/or produced by the compiler.

`-trapuv`

This option has been replaced by the `-DEBUG:trap_uninitialized` option.

`-U`*name*

Undefines a variable for the source preprocessor.

`-use_command`

Uses the command name to determine which compiler to invoke for recognized source files.

`-use_readonly_const`

Puts string literals and file-level `const` qualified initialized variables into a `.rodata` section.

`-use_readwrite_const`

Puts string literals and file-level `const` qualified initialized variables into a readable and writable data section.

`-use_suffix`

Use the file suffix to determine which compiler to invoke.

`-v`

This option has the same functionality as `-show`.

`-version`

Writes compiler release version information to `stdout`. No input file needs to be specified when this option is used.

`-w`[]

Suppresses messages.

`-w2`[]

Counts warnings as errors.

`-Wl,`*opt*[`,`*arg*][`,`*opt*[`,`*arg*]]`...`

Specifies options to be passed directly to the linker.

`-woff`*numberlist*

Specifies message numbers to suppress.

`-woff all`

Suppresses warning messages.

`-x` *lang*

Specifies that the following files are of the specified language, regardless of suffix. Valid values for *lang* are `c`, `c++`, `f`, `f90`, `assembler`, `object`, or `none`.

`-Xcpluscomm`*numberlist*

Applies C++ style comment rules.

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`.

Compiling and Linking

The two compilation processes in Figure 2-1, page 23, 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, as shown in the following examples:

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

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

On the right is the O32 compilation process, invoked by using `-o32`, as shown in the following example:

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

The following steps further describe the compilation stages in Figure 2-1, page 23:

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.
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.

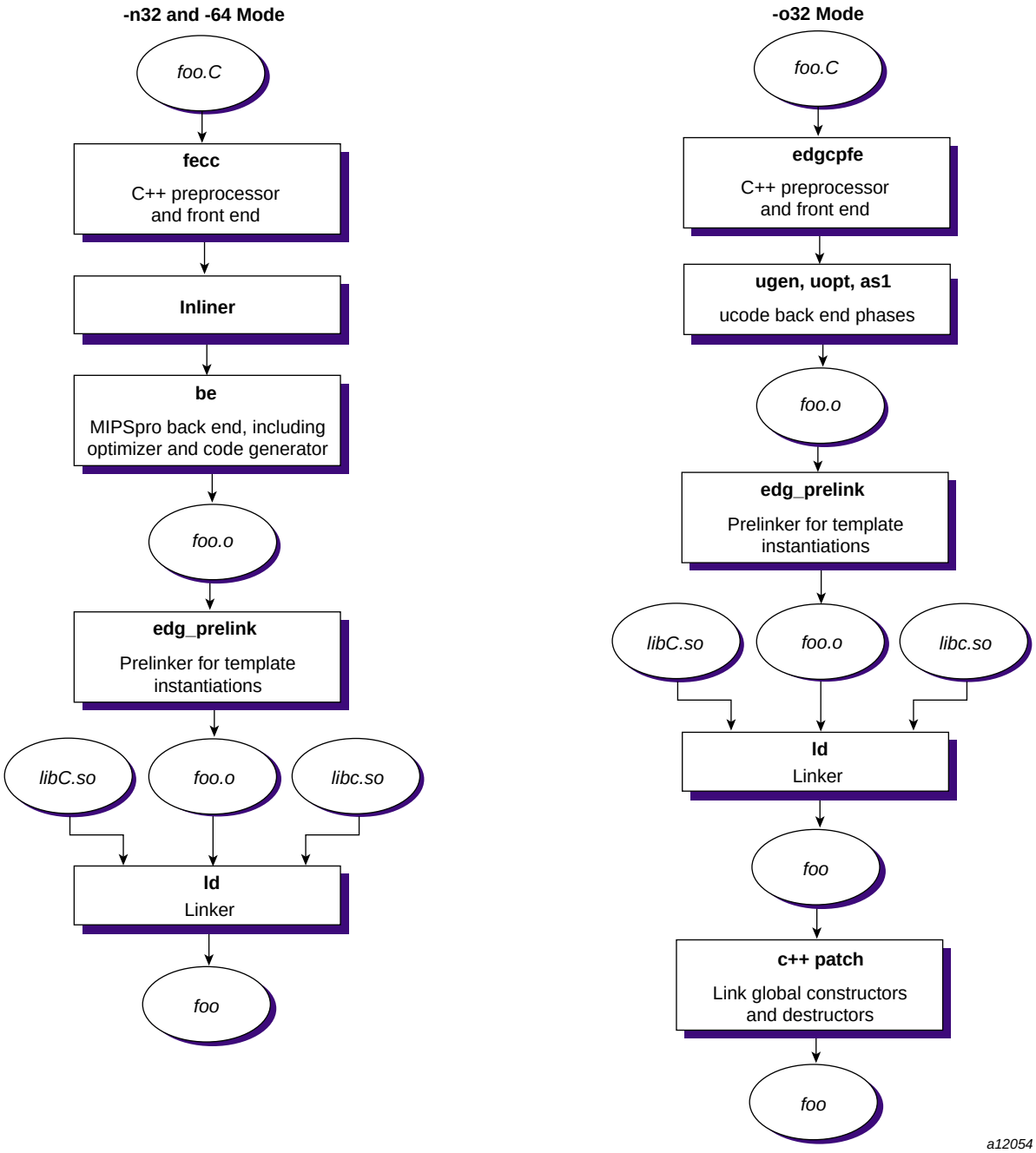

Figure 2-1 The N32, 64, and O32 C++ Compilation Processes

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 ...
  ```

Multilanguage 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, use the following command line:

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

For more information on C++ libraries, see "C++ Libraries", page 7.

Diagnostic Messages

Diagnostic messages can be issued from the C++ front end, the language-independent back end, or the linker.

The front end issues *errors*, *warnings*, and *remarks*. The following examples demonstrate these types of messages.

Example 2-1 Compiler front end ERROR message

The following code fragment produces an error message when compiled with the command shown:

```
void foo()
 {
  x=1;
 }
```

```
% CC -c ex1.cxx
```

The following error message is displayed:

```
cc-1020 CC: ERROR File=ex1.cxx, Line=3
  The identifier "x" is undefined.

  x = 1;
  ^
```

Example 2-2 Compiler front end WARNING message

The following code fragment produces a warning message when compiled with the command shown:

```
int foo(int x)
{
  if (x==1) return 0;
}
```

```
% CC -c ex2.cxx
```

The following warning message is displayed:

```
cc-3604 CC: WARNING File = ex2.cxx, Line = 4
  missing return statement at end of non-void function "foo"

  }
  ^
```

Example 2-3 Compiler front end Remark

The following code fragment produces a remark when compiled with the command shown:

```
class C {};

class D {
  friend C;
};
```

```
% CC -c -fullwarn ex3.cxx
```

The following remark is displayed:

```
Omission of "class" is nonstandard
  friend C;
         ^
```

Remarks occur only if the `-fullwarn` option is specified or if their severity is adjusted, as described in "Adjusting the Severity of Messages", page 28.

Programs with only warnings and remarks will compile; programs with errors will not.

For some diagnostic messages, explanatory information is available by using the `explain`(1) command. In Example 2-1, page 25, the following explanation is provided when the `explain` command shown is used:

```
% explain cc-1020

The identifier %sq is undefined.

The identifier "name" does not appear in a declaration. Check your
declarations and the headers you have included to make sure the
identifier is included. If it is a user defined type or function, it
must be declared before it can be used.
```

Diagnostics from the back end seldom occur.

Diagnostics from the linker are common and are usually clear. Linker messages do not have numbers and the `explain` command cannot be used. One class of linker error is specific to C++ and may be confusing, as the following example demonstrates.

Example 2-4 C++ Linker error

Compiling the following fragment with the indicated command line produces the resulting error:

```
class Base
{
      public:
      virtual void Action ( void );
};
class Derived : public Base
{
      public:
      void Action( void );
};
int main( int ncmds, char *cmds[] )
{
      Derived *it = new Derived;
      it->Action();
}

void Derived::Action( void )
{
}
```

```
% CC ex5.cxx

ld32: ERROR  33 : Unresolved data symbol "Base::_vtbl" -- 1st referenced by ex5.o.
  Use linker option -v to see when and which objects, archives, and dsos are loaded.
ld32: ERROR 33 : Unresolved data symbol "__T_4Base" -- 1st referenced by ex5.o.
  Use linker option -v to see when and which objects, archives, and dsos are loaded.
```

In this example, the unresolved symbols are hard to correlate to anything in the user program. `Base::__vtbl` refers to the virtual table for the class `Base`. Virtual tables are an implementation mechanism for virtual functions. They are data and they need to be in the executable. To avoid duplicating the virtual table for a class in every object file containing the class declaration, the compiler places the virtual table in the class that contains the definition of the first non-inline virtual function in the class.

In the example, the first such function in the class `Base` is Action, and it is not defined anywhere. This problem cannot be detected until link time, and it results in the first diagnostic above. The second diagnostic is related and never appears in the absence of the `__vtbl` message.

If you encounter such a message, look for the first non-inline virtual function in the indicated class and make sure that it is defined somewhere.

Adjusting the Severity of Messages

Users can adjust the severity of diagnostic messages by using the `-diag_error`, `-diag_warning`, and `-diag_remark` options to the compiler. For example, if you do not want a program to compile if there is a missing `return` statement, use the `-diag_error 3604` option. Compiling Example 2-2, page 25, with the command shown below produces the following error:

```
% CC -c -diag_error 3604 ex3.cxx

cc-3604 CC: ERROR File = ex2.cxx, Line = 4
  missing return statement at end of non-void function "foo"
  }
  ^
```

To avoid producing even warnings for this example, use either the `-woff 3604` or `-diag_remark 3604` options.

To see a warning for Example 2-3, page 26, without compiling with the `-fullwarn` option, use the `-diag_remark 3604` option.

It is sometimes possible to reduce an error to a warning or to a remark. These errors are *discretionary errors*. The error in Example 2-1, page 25, is not discretionary, so if `-diag_warning 1020` is specified, the error still occurs. If `-diag_warning 1238` is used, the error in the following example is reduced to a warning:

```
class C {
   int data;
```

```
public:
  C(): data(1) {}
};
int main ()
{
  C c;
  return c.data;
}

cc-1238: ERROR File = ex4.cxx, Line = 10
  The member "C::data" is inaccessible.

  return c.data;
         ^
```

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.
- `elfdump`: the `elfdump` tool lists the contents of an ELF (Executable and Linking 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 N32/64 Compiling and Performance Tuning Guide*.

MIPSpro C++ and the C++ Standard

The MIPSpro C++ compiler implements the C++ language as defined in the C++ International Standard with the exception of the features listed in "Unimplemented C++ Standard Language Features", page 31.

This chapter discusses the following topics:

- "Unimplemented C++ Standard Language Features", page 31, contains a list of standard features not yet supported by the MIPSpro C++ compilers.
- "Feature Control", page 32, contains a list of options that controls the features which are supported.
- "Extensions Accepted", page 33, contains a list of the extensions that are accepted by the MIPSpro compilers, either by default or under command line control.

The compiler provides special compatability modes for compiling older, outmoded programs. See Appendix B, "Cfront Compatability", page 77, and Appendix C, "Anachronisms Accepted", page 83, for information.

Unimplemented C++ Standard Language Features

The MIPSpro C++ compiler implements the full C++ language as described in the *ISO/IEC 14882: Programming Language — C++* document (referred to as "The Standard" in this text) with the following exceptions:

- `extern` inline functions are not supported. Inline functions always have internal linkage. See section 3.5 of the Standard.
- The `export` keyword is not supported. See section 2.11 of the Standard.
- By default, the new template name resolution rules are not supported. See section 14.6 of the Standard. To use the new rules, you can use the `-LANG:do-dependent-name-processing` option.

  Standard header files, like `iostream`, have not been corrected to conform with these rules and will not compile with this option.
- The stack is not unwound on a violated exception specification. See section 15.5.2 of the Standard.

- Placement delete is not implemented (Section 5.3.4).

Feature Control

A number of options in the `LANG` group can be used to control the set of C++ features that are supported. These options can be used, for example, to compile code that does not conform with the Standard in one way or another. It may not always be possible, however, to link together object files, some of which have been compiled with a feature enabled and others with it disabled.

- `-LANG:bool=off`: removes the `bool` keyword. Use this, for example, to compile code in which `bool` is a `typedef` for some integral type.
- `-LANG:exceptions=off`: disables exception handling. `try`, `throw`, `catch`, for example, will not be recognized as keywords. You may want to use this option to save space; some code may run slightly faster with exceptions disabled.
- `-LANG:wchar_t=off`: removes the `wchar_t` keyword. Use this, for example, to compile code in which `wchar_t` is a `typedef` for some integral type.
- `-LANG:namespaces=off`: removes the `namespace` keyword.
- `-LANG:ansi-for-init-scope=off`: affects the scope of the loop variable in the construct:

  ```
  for (int i = 0;...)
  ```

  The Standard specifies that the scope of the loop variable extends only to the end of the `for` statement. The old behavior made the scope extend to the end of the enclosing block. Use this option to revert to that behavior. It is safe to mix object files compiled with and without this option.
- `-LANG:std=off`: disables a number of requirements of the Standard, mostly rather minor and technical. The most important of these are the following:
  - the `typename` keyword is not always required.
  - the template<> syntax is not required for template specializations.
  - the scope of `for` loop variables extends to the enclosing block, just as if `-LANG:ansi-for-init-scope=off` had been specified.

  - there are slight differences in overload resolution, with the effect that some function calls that would be ambiguous under the Standard are resolved by "late tiebreaker" rules.
  - string literals have type `char[]` instead of `const char[]`.

- `-LANG:libc_in_namespace_std=off`: This disables the predefined macro `_LIBC_IN_NAMESPACE_STD`, used for header-file functionality that puts C library functions like `printf` into namespace `std` as required by the C++ Standard. Use this option if you use C header files such as `stdio.h` instead of the C++ versions such as `cstdio`, for example.

Extensions Accepted

- The `long long` and `unsigned long long` types provide a 64-bit integral type. This is accepted in default mode. The use of `long long` is flagged with a warning if the `-ansiW` or `-ansiE` option is specified.
- C99-style variable length arrays are supported with the `-LANG:vla=on` command line option. Not accepted in default mode.
- The `restrict` keyword is supported with the `-LANG:restrict=on` command line option. Not accepted in default mode.

Note that `long long` is accepted by default; `restrict` and variable length arrays are accepted only with special command line options. When you use the special options for `restrict` and variable length arrays, there are no diagnostics for their use even if you specify the `-ansiE` or `-ansiW` options. `long long` is accepted by default and there is a warning if you specify `-ansiE` or `-ansiW`, but never an error.

Using Templates

The information in this chapter is derived from the *C++ Front End Internal Documentation, Version 2.45*, copyright 1992–2000, by the Edison Design Group. Used by the permission of the authors.

Template Instantiation

The C++ programming language includes the concept of *templates*. A template is a description of a class or function that is a model for a family of related classes or functions. Because templates are descriptions of entities (typically, classes) that are parameterizable according to the types they operate upon, they are sometimes called *parameterized types*. For example, you can write a template for a `Stack` class and then use a stack of integers, a stack of floats, and a stack of a user-defined type. In the source code, these might be written as follows:

```
Stack<int>
Stack<float>
Stack<X>
```

From a single source description of the template for a stack, the compiler can create an *instantiation* of the template for each of the types that is required.

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 an external linkage.)
- You can 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 provides 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. Therefore, the compiler cannot do the instantiation automatically in any source file that references it.

- You cannot 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. If more explicit control is needed, instantiation modes and instantiation pragmas are also provided and can be used to provide fine-grained control over the instantiation process.

The following subsections discuss methods of instantiation and other associated topics.

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 the definition of the template entity, a reference that causes the instantiation, and declarations of any types required for the particular instantiation.

Meeting Instantiation Requirements

You can meet the instantiation requirements in several ways:

- You can have each header file that declares a template entity contain either the definition of the entity or another file that contains the definition.
- When the compiler 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. See "Implicit Inclusion", page 47 for more information.

- 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.

Automatic Instantiation Method

The following steps outline the general process for using automatic instantiation.

1. The first time the source files of a program are compiled, no template entities are instantiated. However, template information files (with a default `.ti` suffix) are generated and contain information about things that could have been instantiated in each compilation.
2. When the object files are linked together, 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 original compilation command-line options (saved in the information file) are used for the recompilation.
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). The compiler also receives a definition list file, which lists all the instantiations for which definitions already exist in the set of object files. If during compilation the compiler has the opportunity to instantiate a reference entity that is not on that list, it goes ahead and does the instantiation. It passes back to the prelinker (in the definition list file) a list of the instantiations that it has 'adopted' in this way, so the prelinker can assign them to the file. This 'adoption' process

allows rapid instantiation and assignment of instantiations referenced from new instantiations, and reduces the need to recompile a given file more than once during the prelinking process.

6. The prelinker repeats steps 3–5 until there are no more instantiations to be adjusted.
7. The object files are linked.

After 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. Becasue that definition satisfies whatever references there might be to that entity, the prelinker sees no need to request an instantiation of the entity. If you add 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: A<int>::f() assigned to file test.o
C++ prelinker: executing: usr/lib/edg-prelink -c test.c
```

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 as described in the following subsections.

Instantiations are normally generated as part of the object file of the translation unit in which the instantiations are performed. But when 'one instantiation per object' mode is specified, each instantiation is placed in its own object file.
One-instantiation-per-object mode is useful when generating libraries that need to include copies of the instances referenced from the library. If each instance is not placed in its own object file, it may be impossible to link the library with another library containing some of the same instances. Without this feature it is necessary to create each individual instantiation object file using the manual instantiation mechanism.

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.

Instantiation Modes

Normally, when a file is compiled, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by a command line option.

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`, or `-ptall`. (Automatic template instantiation should make the use of `-ptused` or `-ptall` unnecessary in most cases.)
- `-prelink` (the default) or `-no_prelink`
- `-auto_include` or `-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 turned 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.

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 "Implicit Inclusion", page 47.)

`-no_auto_include`

Disables implicit inclusion of template implementation files.

`-ptv`

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

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

Command Line Instantiation Examples

This section provides you with typical combinations of command line instantiation options, along with an explanation of what these combinations do and how they may be used.

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

`-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 run-time 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.

`-no_auto_include`

Use this option if you are using only 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.

`#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`. See "`#pragma instantiate`".
- `#pragma do_not_instantiate`. See "`#pragma can_instantiate`", page 45.
- `#pragma can_instantiate`. See "`#pragma can_instantiate`", page 45.

#pragma instantiate

The #pragma instantiate directive causes a specific instance of a template declaration to be immediately instantiated.

The syntax of the #pragma instantiate directive is as follows:

```
#pragma instantiate  entity
```

The *entity* argument can be any of the following:

A template class name

```
A<int>
```

A member function name

```
A<int>::foo
```

A member function declaration

```
void A<int>::foo(int, char)
```

A static data member name

```
A<int>::name
```

A template function declaration

```
char* foo(int, float)
```

The template definition of *entity* must be present in the compilation for an instantiation to occur. If you use #pragma instantiate to explicitly request the instantiation of a class or function for which no template definition is available, the compiler issues a warning.

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

The argument to an instantiation #pragma directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

A member function name (for example, A<int>::foo) can be used as an argument for a #pragma instantiate directive only if it refers to a single, user-defined member function that is not an overloaded function. Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a

compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as the following example shows:

```
char * A<int>::foo(int))
```

Note: Using the `#pragma instantiate` directive to instantiate a template class is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class, you can exclude a given member function or static data member by using the `#pragma do_not_instantiate` directive.

#pragma can_instantiate

The `#pragma can_instantiate` directive indicates that the specified entity can be instantiated in the current compilation, but need not be. It is used in conjunction with automatic instantiation to indicate potential sites for instantiation if the template entity is deemed to be required by the compiler.

The syntax of the `#pragma can_instantiate` directive is as follows:

```
#pragma can_instantiate entity
```

The argument, *entity*, can be any of the following:

A template class name

```
A<int>
```

A member function name

```
A<int>::foo
```

A member function declaration

```
void A<int>::foo(int, char)
```

A static data member name

```
A<int>::name
```

A template function declaration

```
char* foo(int, float)
```

The template definition of *entity* must be present in the compilation for an instantiation to occur. If you use `#pragma can_instantiate` to explicitly request the instantiation of a class or function for which no template definition is available, the compiler issues a warning.

The argument to a `#pragma can_instantiate` directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

A member function name (for example, `A<int>::foo`) can be used as an argument for a `#pragma can_instantiate` directive only if it refers to a single, user-defined member function that is not an overloaded function. Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as shown in the following example:

```
char * A<int>::foo(int)
```

#pragma do_not_instantiate

The `#pragma do_not_instantiate` directive suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition is supplied.

The syntax of the `#pragma do_not_instantiate` directive is as follows:

```
#pragma do_not_instantiate entity
```

The argument, *entity*, can be any of the following:

A template class name

```
A<int>
```

A member function name

```
A<int>::foo
```

A member function declaration

```
void A<int>::foo(int, char)
```

A static data member name

```
A<int>::name
```

A template function declaration

```
char* foo(int, float)
```

The argument to a `#pragma do_not_instantiate` directive cannot be a compiler-generated function, an inline function, or a pure virtual function.

A member function name (for example, `A<int>::foo`) can be used as an argument for the `#pragma do_not_instantiate` directive only if it refers to a single, user-defined member function that is not overloaded. Compiler-generated functions are not considered, so a name can refer to a user-defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be specified by providing the complete member function declaration, as the following example shows:

```
char * A<int>::foo(int)
```

Implicit Inclusion

When implicit inclusion is enabled, the compiler assumes that if it needs a definition to instatiate a template entity declared in a `.h` file, it can implicitly include the corresponding `.C` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, the compiler looks to see if a file `xyz.C` exists, and if so it will process it as if it were included at the end of the main source file.

To find the template definition file for a given template entity the compiler needs to know the full path name of the file in which the template was declared and whether the file was included using the system include syntax (that is, `#include <file.h>`). This information is not available for preprocessed source containing `#line` directives. Therefore, the compiler does not attempt implicit inclusion for source code containing `#line` directives.

The definition-file suffixes that are tried are the following:

```
.c
.C
.cpp
.CPP
.cxx
.CXX
.cc
.CC
.c++
.C++
```

Implicit inclusion works well with 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 inclusions are only performed during the normal compilation of a file (that is, not when doing only preprocessing). A common means of investigating certain kinds of problems is to produce a preprocessed source file that can be inspected. When using implicit inclusion it is sometimes desirable for the preprocessed source file to include any implicitly included files. This may be done using the `-FE:generate_preprocessed_output` command line option. This causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files will appear as part of the preprocessed output in the precise location at which they were included in the compilation.

The Auto-Parallelizing Option (APO)

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.

C/C++ Command Line Options That Affect APO

Several `cc`(1) and `CC`(1) command line options control APO's effect on your program. For example, the following command line invokes APO and requests aggressive optimization:

```
CC -apo -O3 zebra.c
```

The following subsections describe the effects that various C/C++ command line options have on APO.

Note: If you invoke the loader separately, you must specify the `-apo` option on the `ld`(1) command line.

`-apo`

The `-apo` option invokes APO. When this option is enabled, the compiler automatically converts sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so. Specifying `-apo` also enables the `-mp` option, which enables recognition of the parallel directives inserted into your code.

`-apokeep` and `-apolist`

The `-apokeep` and `-apolist` options control output files. Both options generate *file*`.list`, which is a listing file that contains information on the loops that were parallelized and explains why others were not parallelized.

When `-apokeep` is specified, the compiler writes *file*`.list`, and in addition, it retains *file*`.anl` and *file*`.m`. The ProMP tools use *file*`.anl`. For more information on ProMP, see the *ProDev ProMP User's Guide*. *file*`.m` is an annotated version of your source code that shows the insertion of multiprocessing directives.

When `-IPA` is specified with the `-apokeep` option, the default settings for IPA suboptions are used, with the exception of `-IPA:inline`, which is set to `OFF`.

For more information on the content of *file*`.list`, *file*`.anl`, and *file*`.m`, see "Files", page 54.

Note: Because of data conflicts, do not specify the `-mplist` or `-CLIST` options when `-apokeep` is specified.

-CLIST:...

This option generates a C/C++ listing and directs the compiler to write an equivalent parallelized program in *file*`.w2c.c`. For more information on the content of *file*`.w2c.c`, see "Files", page 54.

-IPA:...

Interprocedural analysis (IPA) is invoked by the `-ipa` or `-IPA` command line option. It performs program optimizations that can only be done by examining the whole program, not parts of a program.

When APO is invoked with IPA, only those loops whose function calls were determined to be safe by the APO are optimized.

If IPA expands functions inline in a calling routine, the functions are compiled with the options of the calling routine. If the calling routine is not compiled with `-apo`, none of its inlined functions are parallelized. This is true even if the functions are compiled separately with `-apo` because with IPA, automatic parallelization is deferred until link time.

When `-apokeep` or `-pcakeep` are specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption, which is set to `OFF`.

For more information on the effect of IPA, see "Loops Containing Function Calls", page 66. For more information on IPA itself, see the `ipa`(5) man page.

-LNO:...

The `-LNO` options control the Loop Nest Optimizer (LNO). LNO performs loop optimizations that better exploit caches and instruction-level parallelism. The following LNO options are of particular interest to APO users:

- `-LNO:auto_dist=on`. This option requests that APO insert data distribution directives to provide the best memory utilization on Origin 2000 systems.
- `-LNO:ignore_pragmas=`*setting*. This option directs APO to ignore all of the directives and assertions described in "Compiler Directives", page 58.
- `-LNO:parallel_overhead=`*num_cycles*. This option allows you to override certain compiler assumptions regarding the efficiency to be gained by executing

certain loops in parallel rather than serially. Specifically, changing this setting changes the default estimate of the cost to invoke a parallel loop in your run-time environment. This estimate varies depending on your particular run-time environment, but it is typically several thousand machine cycles.

You can view the transformed code in the original source language after LNO performs its transformations. Two translators, integrated into the compiler, convert the compiler's internal representation into the original source language. You can invoke the desired translator by using the `CC -CLIST:=on` option. For example, the following command creates an `a.out` object file and the C/C++ file `test.w2c.c`:

```
CC -O3 -CLIST:=on test.c
```

Because it is generated at a later stage of the compilation, this `.w2c.c` file differs somewhat from the `.w2c.c` file generated by the `-apokeep` option (see "`-apokeep` and `-apolist`", page 50). You can read the `.w2c.c` file, which is a compilable C/C++ representation of the original program after the LNO phase. Because the LNO is not a preprocessor, recompiling the *file*`.w2c.c` can result in an executable that differs from the original compilation of the `.c` file.

`-O3`

To obtain maximum performance, specify `-O3` when compiling with APO enabled. The optimization at this level maximizes code quality even if it requires extensive compile time or relaxes the language rules. The `-O3` option uses transformations that are usually beneficial but can sometimes hurt performance. This optimization may cause noticeable changes in floating-point results due to the relaxation of operation-ordering rules. Floating-point optimization is discussed further in "`-OPT:...`", page 52.

`-OPT:...`

The `-OPT` command line option controls general optimizations that are not associated with a distinct compiler phase.

The `-OPT:roundoff=`*n* option controls floating-point accuracy and the behavior of overflow and underflow exceptions relative to the source language rules.

When `-O3` is in effect, the default rounding setting is `-OPT:roundoff=2`. This setting allows transformations with extensive effects on floating-point results. It allows associative rearrangement across loop iterations and the distribution of

multiplication over addition and subtraction. It disallows only transformations known to cause overflow, underflow, or cumulative round-off errors for a wide range of floating-point operands.

At `-OPT:roundoff=2` or `3`, APO can change the sequence of a loop's floating-point operations in order to parallelize it. Because floating-point operations have finite precision, this change can cause slightly different results. If you want to avoid these differences by not having such loops parallelized, you must compile with `-OPT:roundoff=0` or `-OPT:roundoff=1`.

Example. APO parallelizes the following loop when compiled with the default settings of `-OPT:roundoff=2` and `-O3`:

```
float a, b[100];
for(i=0; i<100; i++)
    a = a + b[i];
```

At the start of the loop, each processor gets a private copy of `a` in which to hold a partial sum. At the end of the loop, the partial sum in each processor's copy is added to the total in the original, global copy. This value of `a` can be different from the value generated by a version of the loop that is not parallelized.

`-pca`, `-pcakeep`, `-pcalist`

The `-pca` option invokes APO. For the O32 ABI, the `-pca` option invokes Power C. The `-pcakeep` and `-pcalist` options control output files.

When `-IPA` is specified with the `-pcakeep` option, the default settings for IPA suboptions are used, with the exception of `-IPA:inline`, which is set to `OFF`.

Note: These options are outmoded. The preferred way of invoking APO is through the `-apo` option, and the preferred way to obtain a listing is through the `-apolist` option. For more information on these options, see "`-apo`", page 50, and "`-apokeep` and `-apolist`", page 50.

---

file

Your input file.

For information on files used and generated when APO is enabled, see "Files".

Files

APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. You can use these listings to identify constructs that inhibit parallelization. When you remove these constructs, you can often improve program performance dramatically.

When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. To determine where the program spends its execution time, you can use tools such as SpeedShop and the WorkShop ProMP Parallel Analyzer View described in *ProDev WorkShop: ProMP User's Guide*.

The following sections describe the content of the files generated by APO.

The *file*`.list` File

The `-apolist` and `-apokeep` options generate files that list the original loops in the program along with messages indicating if the loops were parallelized. For loops that were not parallelized, an explanation is provided.

Example. The following function resides in file `testl.c`:

```
void sub(double arr[], int n)
{
  extern void foo(double);
  int i;
  for(i=1; i<n; i++)
  {
    arr[i] += arr[i-1];
  }
  for(i=0; i<n; i++)
  {
    arr[i] += 7.0;
    foo(arr[i]);
  }
  for(i=0; i<n; i++)
  {
    arr[i] += 7.0;
  }
}
```

File `testl.c` is compiled with the following command:

```
cc -O3 -n32 -mips4 -apolist -c testl.c
```

APO produces file `testl.list`:

```
Parallelization Log for Subprogram sub
    5: Not Parallel
         Array dependence from arr on line 6 to arr on line 6.

    8: Not Parallel
         Call foo on line 10.

   12: PARALLEL (Auto) __mpdo_sub1
```

Note the message for line 12. Whenever a loop is run in parallel, the parallel version of the loop is put in its own function. The MIPSpro profiling tools attribute all the time spent in the loop to this function. The last line indicates that the name of the function is `__mpdo_sub1`.

The *file*`.w2f.c` File

File *file*`.w2c.c` contains code that mimics the behavior of programs after they undergo automatic parallelization. The representation is designed to be readable so that you can see what portions of the original code were not parallelized. You can use this information to change the original program.

The compiler creates *file*`.w2c.c` by invoking the appropriate translator to turn the compiler's internal representations into C/C++. In most cases, the files contain valid code that can be recompiled, although compiling *file*`.w2c.c` without APO enabled does not produce object code that is exactly the same as that generated when APO is enabled on the original source.

The `-apolist` option generates *file*`.w2c.c`. Because it is generated at an earlier stage of the compilation, *file*`.w2c.c` from `-apolist` is more easily understood than *file*`.w2c.c` generated from `-CLIST:=on` option. On the other hand, the `-CLIST` option shows more of the optimizations that were performed. The parallelized program in *file*`.w2c.c` uses OpenMP directives.

Example. File `testw2.c` is compiled with the following command:

```
cc -O3 -n32 -mips4 -c -apo -apolist -c testw2.c
```

```
void trivial(float a[])
{
    int i;
    for(i=0; i<10000; i++) {
      a[i] = 0.0;
    }
}
```

Compiling `testw2.c` generates an object file, `testw2.o`, and listing file `testw2.w2c.c`, which contains the following code:

```
/*****************************************************
 * C file translated from WHIRL Wed Oct 28 14:03:23 1998
 *****************************************************/
/* Include file-level type and variable decls */
#include "testw2.w2c.h"


void trivial(
  _IEEE32(*a0)[])
{
  register _INT32 i0;

  /* PARALLEL DO will be converted to SUBROUTINE __mpdo_trivial1 */;
#pragma parallel
  {
#pragma pfor
#pragma local(i0)
#pragma shared(a0)
    for(i0 = 0; i0 <= 9999; i0 = i0 + 1)
    {
      (*a0)[i0] = 0.0F;
    }
  }
  return;
} /* trivial */
```

Note: `WHIRL` is the name for the compiler's intermediate representation.

As explained in "The *file*`.list` File", page 54, parallel versions of loops are put in their own functions. In this example, that function is `__mpdo_trivial_1`. `#pragma omp parallel` is an OpenMP directive that specifies a parallel region containing a single `DO` directive.

About the `.m` and `.anl` Files

The `-apokeep` option generates *file*`.list`. It also generates *file*`.m` and *file*`.anl`, which are used by Workshop ProMP.

file`.m` is similar to the *file*`.w2c.c` file but is more like original source code; it is based on OpenMP and mimics the behavior of the program after automatic parallelization.

WorkShop ProMP is a Silicon Graphics product that provides a graphical interface to aid in both automatic and manual parallelization for C/C++. The WorkShop ProMP Parallel Analyzer View helps you understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. For more information, see the *ProDev WorkShop: ProMP User's Guide* and the *ProDev WorkShop: Performance Analyzer User's Guide*.

SpeedShop, another Silicon Graphics product, allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop includes a set of commands and a number of libraries to support the commands. For more information, see the *SpeedShop User's Guide*.

Running Your Program

Running a parallelized version of your program is no different from running a sequential one. The same binary output file can be executed on various numbers of processors. The default is to have the run-time environment select the number of processors to use based on how many are available.

You can change the default behavior by setting the `OMP_NUM_THREADS` environment variable, which tells the system to use an explicit number of processors. The following statement causes the program to create two threads regardless of the number of processors available:

```
setenv OMP_NUM_THREADS 2
```

The OMP_DYNAMIC environment variable allows you to control whether the run-time environment should dynamically adjust the number of threads available for executing parallel regions to optimize system resources. The default value is ON. If OMP_DYNAMIC is set to OFF, dynamic adjustment is disabled.

For more information on these and other environment variables, see the pe_environ(5) man page.

Compiler Directives

APO works in conjunction with the OpenMP C/C++ API directives and with the Origin series directives. You can use these directives to manually parallelize some loop nests, while leaving others to APO. This approach has the following positive and negative aspects:

- As a positive aspect, the OpenMP and Origin series directives are well defined and deterministic. If you use a directive, the specified loop is run in parallel. This assumes that the trip count is greater than one and that the specified loop is not nested in another parallel loop.
- The negative side to this is that you must carefully analyze the code to determine that parallelism is safe. In particular, you may need to specify special attributes for some variables, such as private or reduction, or specify explicit synchronizations, such as a barrier or a critical section.

In addition to the OpenMP and Origin series directives, you can also use the APO-specific directives described in this section. These directives give APO more information about your code.

Note: APO also recognizes the Silicon Graphics multiprocessing directives. These directives are outmoded, and you must include the -mp option on the CC(1) command line in order for the compiler to recognize them. The OpenMP directive set is the preferred directive set for multiprocessing.

The APO directives can affect certain optimizations, such as loop interchange, during the compiling process. To direct the compiler to disregard any of the preceding directives, specify the -x*dirlist* option.

The APO directives are as follows:

- `#pragma concurrent call`. This directive directs APO to ignore dependencies in function calls that would inhibit parallelization. For more information on this directive, see "`#pragma concurrent call`", page 59.
- `#pragma concurrent`. This directive asserts that APO should not let perceived dependencies between two references to the same array inhibit parallelizing. For more information on this directive, see "`#pragma concurrent`", page 61.
- `#pragma serial`. This directive requests that the following loop be executed in serial mode. For more information on this directive, see "`#pragma serial`", page 62.
- `#pragma prefer concurrent`. This directive parallelizes the following loop if it is safe. For more information on this directive, see "`#pragma prefer concurrent`", page 62.
- `#pragma permutation (`*array_name*`)`. Asserts that array *array_name* is a permutation array. For more information on this directive, see "`#pragma permutation`", page 63.
- `#pragma no concurrentize` and `#pragma concurrentize`. The `#pragma no concurrentize` directive inhibits either parallelization of all loops in a function or parallelization of all loops in a file. The `#pragma concurrentize` directive overrides the `#pragma no concurrentize` directive, and its effect varies with its placement. For more information on these directives, see "`#pragma no concurrentize`, `#pragma concurrentize`", page 64.

Note: The compiler honors the following APO directives even if the `-apo` option is not included on your command line:

- `#pragma concurrent call`
- `#pragma prefer concurrent`
- `#pragma permutation (`*array_name*`)`

`#pragma concurrent call`

The `#pragma concurrent call` directive instructs APO to ignore the dependencies of function and function calls contained in the loop that follows the

assertion. The directive applies to the loop that immediately follows it and to all loops nested inside that loop. Other points to be aware of are the following:

Note: The directive affects the compilation even when `-apo` is not specified.

APO ignores potential dependencies in function `fred()` when it analyzes the following loop:

```
#pragma concurrent call
for(i=0; i<n; i++) {
    fred();
    ...
}
```

To prevent incorrect parallelization, make sure the following conditions are met when using `#pragma concurrent call`:

- A function inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.
- A function inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the function.

Example. The following code shows an illegal use of the directive. Function `fred()` writes to variable `x`, which is also read from by `wilma()` during other iterations, and the directive instructs APO to ignore this dependence.

```
void fred(float *b, int i, float *t)
{
  *t = b[i];
}
void wilma(float *a, int i, float *t)
{
  a[i] = *t;
}

#pragma concurrent call
for(i=0; i<m; i++)
{
  fred(b, i, &x);
```

```
  wilma(a, i, &x);
}
```

The following example shows how you can manually parallelize the preceding example safely by 'localizing' variable `x` with a declaration `float x;` at the top of the loop body.

```
#pragma concurrent call
for (i=0, i<m, i++) {
   float x;
   fred(b, i, &x);
   wilma(a,i, &x);
}
```

#pragma concurrent

The `#pragma concurrent` directive instructs APO, when analyzing the loop immediately following this directive, to ignore all dependencies between two references to the same array. If there are real dependencies between array references, the `#pragma concurrent` directive can cause APO to generate incorrect code.

Note: This directive affects the compilation even when `-apo` is not specified.

The following example shows correct use of this directive when `m > n`:

```
#pragma concurrent
for(i=0; i<n; i++)
    a[i] = a[i+m];
```

Be aware of the following points when using this directive:

- If multiple loops in a nest can be parallelized, `#pragma concurrent` causes APO to parallelize the loop immediately following the assertion.
- Applying this directive to an inner loop can cause the loop to be made outermost by APO's loop interchange operations.
- This directive does not affect how APO analyzes function calls. For more information on APO's interaction with function calls, see "`#pragma concurrent call`", page 59.

- This directive does not affect how APO analyzes dependencies between two potentially aliased pointers.
- The compiler may find some obvious real dependencies. If it does so, it ignores this directive.

#pragma serial

The `#pragma serial` instructs APO not to parallelize the loop following the assertion; the loop is executed in serial mode. APO can, however, parallelize another loop in the same nest. The parallelized loop can be either inside or outside the designated sequential loop.

Example. The following code fragment contains a directive that requests that loop `j` be run serially:

```
for(i=0; i<m; i++) {
    #pragma serial
    for(j=0; j<n; j++)
        a[i][j] = b[i][j];
    ...
}
```

The directive applies only to the loop that immediately follows it. For example, APO still tries to parallelize loop `i`. This directive is useful in cases like this when the value of `n` is known to be very small.

#pragma prefer concurrent

The `#pragma prefer concurrent` directive instructs APO to parallelize the loop immediately following the directive if it is safe to do so.

Example. The following code fragment encourages APO to run loop `i` in parallel:

```
#pragma prefer concurrent
for(i=0; i<m; i++) {
    for(j=0; j<n; j++)
        a[i][j] = b[i][j];
    ...
}
```

When dealing with nested loops, APO follows these guidelines:

- If the loop specified by the `#pragma prefer concurrent` directive is safe to parallelize, APO parallelizes the specified loop even if other loops in the nest are safe.
- If the specified loop is not safe to parallelize, APO parallelizes a different loop that is safe.
- If this directive is applied to an inner loop, APO can interchange the loop and make the specified loop the outermost loop.
- If this directive is applied to more than one loop in a nest, APO parallelizes one of the specified loops.

`#pragma permutation`

When placed inside a function, the `#pragma permutation (`*array_name*`)` directive informs APO that *array_name* is a *permutation* array. A permutation array is one in which every element of the array has a distinct value.

The directive does not require the permutation array to be *dense*. That is, within the array, every `b[i]` must have a distinct value, but there can be gaps between the values, such as `b[1]` = 1, `b[2]` = 4, `b[3]` = 9, and so on.

Note: This directive affects compilation even when `-apo` is not specified.

Example. In the following code fragment, array `b` is declared to be a permutation array for both loops in `sub1()`:

```
void sub1(int n)
{
  int i;
  extern int a[], b[];
  for(i=0; i<n; i++)
  {
    a[b[i]] = i;
  }
  #pragma permutation (b)
  for(i=0; i<n; i++)
```

```
  {
    a[b[i]] = i;
  }
}
```

Note the following points about this directive:

- As shown in the example, you can use this directive to parallelize loops that use arrays for indirect addressing. Without this directive, APO cannot determine that the array elements used as indexes are distinct.
- `#pragma permutation (`*array_name*`)` affects every loop in a function, even those that appear before it.

`#pragma no concurrentize`, `#pragma concurrentize`

The `#pragma no concurrentize` directive inhibits parallelization. Its effect depends on its placement.

- When placed inside functions, this directive inhibits parallelization. In the following example, no loops inside `sub1()` are parallelized:

```
void sub1() {
#pragma no concurrentize
    ...
}
```

- When placed outside of a function, `#pragma no concurrentize` prevents the parallelization of all functions in the file, even those that appear ahead of it in the file. Loops inside functions `sub2()` and `sub3()` are not parallelized in the following example:

```
void sub2()
{
  ...
}
#pragma no concurrentize
void sub3()
{
  ...
}
```

The `#pragma concurrentize` directive, when placed inside a function, overrides a `#pragma no concurrentize` directive that is placed outside of it. Thus, this directive allows you to selectively parallelize functions in a file that has been made sequential with a `#pragma no concurrentize` directive.

Troubleshooting Incomplete Optimizations

Some loops cannot be safely parallelized and others are written in ways that inhibit APO's efficiency. The following subsections describe the steps you can take to make APO more effective. The sections that follow, and the topics they discuss, are as follows:

- "Constructs That Inhibit Parallelization", page 65, describes constructs that inhibit parallelization.
- "Constructs That Reduce Performance of Parallelized Code", page 69, describes constructs that reduce performance of parallelized code.

Constructs That Inhibit Parallelization

A program's performance can be severely constrained if APO cannot recognize that a loop is safe to parallelize. APO analyzes every loop in a program. If a loop does not appear safe, it does not parallelize that loop. The following sections describe constructs that can inhibit parallelization:

- "Loops Containing Data Dependencies", page 66, describes basic data dependencies.
- "Loops Containing Function Calls", page 66, describes function calls.
- "Loops Containing `goto` Statements", page 66, describes `goto` statements.
- "Loops Containing Problematic Array Constructs", page 66, describes problematic array subscripts.
- "Loops Containing Local Variables", page 68, describes conditionally assigned local variables.

In many instances, loops containing the previous constructs can be parallelized after minor changes. Reviewing the information generated in program *file*`.list`, described in "The *file*`.list` File", page 54, can show you if any of these constructs are in your code.

Loops Containing Data Dependencies

Generally, a loop is safe if there are no data dependencies, such as a variable being assigned in one iteration of a loop and used in another. APO does not parallelize loops for which it detects data dependencies.

Loops Containing Function Calls

By default, APO does not parallelize a loop that contains a function call because the function in one iteration of the loop can modify or depend on data in other iterations.

You can, however, use interprocedural analysis (IPA) to provide the MIPSpro APO with enough information to parallelize some loops containing function calls. IPA is specified by the `-ipa` command line option. For more information on IPA, see `ipa`(5) and the *MIPSpro N32/64 Compiling and Performance Tuning Guide*.

You can also direct APO to ignore function call dependencies when analyzing the specified loops by using the `#pragma concurrent call` directive described in "`#pragma concurrent call`", page 59.

Loops Containing `goto` Statements

A `goto` statement is an unstructured control flow. APO converts most unstructured control flows in loops into structured flows that can be parallelized. However, `goto` statements in loops can still cause the following problems:

- Unstructured control flows. APO is unable to restructure all types of flow control in loops. You must either restructure these control flows or manually parallelize the loops containing them.
- Early exits from loops. Loops with early exits cannot be parallelized, either automatically or manually.

For improved performance, remove `goto` statements from loops to be considered candidates for parallelization.

Loops Containing Problematic Array Constructs

The following array constructs inhibit parallelization and should be removed whenever APO is used:

- Arrays with subscripts that are indirect array references. APO cannot analyze indirect array references. The following loop cannot be run safely in parallel if the indirect reference `b[i]` is equal to the same value for different iterations of `i`:

```
for(i=0; i<n; i++)
    a[b[i]] = ...
```

  If every element of array `b` is unique, the loop can safely be made parallel. To achieve automatic parallelism in such cases, use the `#pragma permutation(b)` directive, as discussed in "`#pragma permutation`", page 63.

- Arrays with unanalyzable subscripts. APO cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain the following elements:

  – Literal constants (`1`, `2`, `3`, …)

  – Variables (`i`, `j`, `k`, …)

  – The product of a literal constant and a variable, such as `n*5` or `k*32`

  – A sum or difference of any combination of the first three items, such as `n*21+k-251`

  In the following example, APO cannot analyze the division operator (`/`) in the array subscript and cannot reorder the loop:

```
for(i=0; i<n; i+=2)
    a[i/2] = ...;
```

- Unknown information. In the following example there may be hidden knowledge about the relationship between variables `m` and `n`:

```
for(i=0; i<n; i++)
    a[i] = a[i+m];
```

  The loop can be run in parallel if `m > n` because the array reference does not overlap. However, APO does not know the value of the variables and therefore cannot make the loop parallel. You can use the `#pragma concurrent` directive to have APO automatically parallelize this loop. For more information on this directive, see "`#pragma concurrent`", page 61.

Loops Containing Local Variables

When parallelizing a loop, APO often localizes (privatizes) temporary scalar and array variables by giving each processor its own non-shared copy of them. In the following example, array `tmp` is used for local scratch space:

```
for(i=0; i<n; i++) {
    for(j=0; j<n; j++)
      tmp[j] = i+j;
    for(j=0; j<n; j++)
      a[i][j] = a[i][j] + tmp[j];
}
```

To successfully parallelize the outer loop (`i`), APO must give each processor a distinct, private copy of array `tmp`. In this example, it is able to localize `tmp` and, thereby, to parallelize the loop.

APO cannot parallelize a loop when a conditionally assigned temporary variable might be used outside of the loop, as in the following example:

```
extern int t;
for(i=0; i<n; i++) {
    if(b[i]) {
        t = ...;
        a[i] += t;
    }
}
s2();
```

If the loop were to be run in parallel, a problem would arise if the value of `t` were used inside function `s2()` because it is not known which processor's private copy of `t` should be used by `s2()`. If `t` were not conditionally assigned, the processor that executed iteration `i == n-1` would be used. Because `t` is conditionally assigned, APO cannot determine which copy to use.

The solution comes with the realization that the loop is inherently parallel if the conditionally assigned variable `t` is localized. If the value of `t` is not used outside the loop, replace `t` with a local variable. Unless `t` is a local variable, APO assumes that `s2()` might use it.

Constructs That Reduce Performance of Parallelized Code

APO parallelizes a loop by distributing its iterations among the available processors. Loop nesting, loops with low trip counts, and other program characteristics can affect the efficiency of APO. The following subsections describe the effect that these and other programming constructs can have on APO's ability to parallelize:

- "Parallelizing Nested Loops", page 69, describes parallelizing nested loops.
- "Parallelizing Loops with Small or Indeterminate Trip Counts", page 70, describes parallelizing loops with small or indeterminate trip counts.
- "Parallelizing Loops with Poor Data Locality", page 71, describes parallelizing loops that exhibit poor data locality.

Parallelizing Nested Loops

APO can parallelize only one loop in a loop nest. In these cases, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from that fact that more processors end up processing larger sections of the program. This saves synchronization and other overhead costs.

Example 1. Consider the following simple loop nest:

```
for(i=0; i<n; i++)
    for(j=0; j<m; j++)
        for(k=0; k<l; k++)
           ...
```

When parallelizing nested loops `i`, `j`, and `k`, APO parallelizes only one of the loops. Effective loop nest parallelization depends on the loop that APO chooses, but it is possible for APO to choose an inferior loop to be parallelized. APO may attempt to interchange loops to make a more promising one the outermost. If the outermost loop attempt fails, APO attempts to parallelize an inner loop.

"The *file*`.list` File", page 54, describes *file*`.list`. This output file contains information that tells you which loop in a nest was parallelized. Because of the potential for improved performance, it is useful for you to modify your code so that the outermost loop is the one parallelized.

For every loop that is parallelized, APO generates a test to determine whether the loop is being called from within either another parallel loop or from within a parallel region. In some cases, you can minimize the extra testing that APO must perform by

inserting directives into your code to inhibit parallelization testing. The following example demonstrates this:

Example 2:

```
void sub(int i, int n) {
   int j;
   #pragma serial
   for(j=0; j<n; j++) {
       ...
   }
}
void caller(int n) {
    int i;
    #pragma concurrent call
    for(i=0; i<n; i++) {
        sub(i, n);
    }
}
```

Assume that `sub()` is called only from within `caller()`. The loop in `caller()` is parallelized, so the loop in `sub()` can never be run in parallel. In this case, the test is avoided by using the `#pragma serial` directive, as shown, to force the sequential execution of the loop.

For more information on this compiler directive, see "`#pragma serial`", page 62.

Parallelizing Loops with Small or Indeterminate Trip Counts

The *trip count* is the number of times a loop is executed. Loops with large trip counts are the best candidates for parallelization. The following paragraphs show how to modify your program if your program contains loops with small trip counts or loops with indeterminate trip counts:

- Loops with small trip counts generally run faster when they are not parallelized. Consider the following loop nest:

```
#pragma prefer serial
for(i=0; i<m; i++) {
  for(j=0; j<n; j++) {
    ...
  }
}
```

Without the directive, APO would attempt to parallelize loop `i` because it is outermost. If `m` is very small, it would be better to interchange the loops and make loop `j` outermost, so that it would be parallelized. If that is not possible, and if APO cannot determine that `m` is small, you can use a `#pragma prefer serial` directive, as shown, to indicate to APO that it is better to parallelize loop `j`.

- Loops with large trip counts run faster if they are unconditionally parallelized. Consider the following loop:

```
#pragma prefer concurrent
for(j=0; j<n; j++)
  ...
```

Without the directive, if the trip count is not known (and sometimes even if it is), APO parallelizes the loop conditionally. It generates code for both a parallel and a sequential version of the loop, plus code to select the version to use, based on the trip count, the code inside the loop's body, the number of processors available, and an estimate of the cost to invoke a parallel loop in that run-time environment.

You can avoid the overhead of conditional parallelization by using the `#pragma prefer concurrent` directive, as shown, to indicate to APO that only the parallel version of the loop should be generated.

Parallelizing Loops with Poor Data Locality

Computer memory has a hierarchical organization. Higher up the hierarchy, memory becomes closer to the CPU, faster, more expensive, and more limited in size. Cache memory is at the top of the hierarchy, and main memory is further down in the hierarchy. In multiprocessor systems, each processor has its own cache memory. Because it is time consuming for one processor to access another processor's cache, a program's performance is best when each processor has the data it needs in its own cache.

Programs, especially those that include extensive looping, often exhibit *locality of reference*, which means that if a memory location is referenced, it is probable that it or a nearby location will be referenced in the near future. Loops designed to take advantage of locality do a better job of concentrating data in memory, increasing the probability that a processor will find the data it needs in its own cache.

The following examples show the effect of locality on parallelization. Assume that the loops are to be parallelized and that there are p processors.

Example 1. Distribution of Iterations.

```
for(i=0; i<n; i++) {
    ...a[i]...
}
for(i=n-1; i>=0; i--) {
    ...a[i]...
}
```

In the first loop, the first processor accesses the first `n/`*p* elements of `a`; the second processor accesses the next `n/`*p* elements; and so on. In the second loop, the distribution of iterations is reversed. That is, the first processor accesses the last `n/`*p* elements of `a`, and so on. Most elements are not in the cache of the processor needing them during the second loop. This code fragment would run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 2. Two Nests in Sequence.

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      a[i][j] = b[j][i] + ...;

for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      b[i][j] = a[j][i] + ...;
```

In example 2, APO may parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor accesses the first `n/`*p* rows of `a` and the first `n/`*p* columns of `b`. In the second nest, the first processor accesses the first `n/`*p* columns of `a` and the first `N/`*p* rows of `B`. This example runs much more efficiently if you parallelize the `i` loop in one nest and the `j` loop in the other. You can instruct APO to do this by inserting a `#pragma prefer serial` directive just prior to the `i` loop that contains the `j` loop that you want to be parallelized.

Language Features Not in the ARM

For a number of years, *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup (the "ARM") functioned as a de facto standard for C++ while the official ANSI standard was under development. This appendix describes the most important differences between standard C++ and C++ as described in the ARM.

The following features are not described in the *Annotated C++ Reference Manual*. They are implemented in C++ as defined by the Standard and are implemented 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.

Cfront Compatability

The SGI C++ compiler 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 "Extensions Accepted in Cfront-Compatibility Mode", page 77, and "Cfront Compatibility Restrictions", page 82, for details.

Extensions Accepted in Cfront-Compatibility Mode

The information in this section is derived from the *C++ Front End Internal Documentation, Version 2.45*, copyright 1992–2000, by the Edison Design Group. Used by permission of the authors.

The following extensions are accepted in Cfront-compatibility mode (by using the `-cfront` option):

- Type qualifiers on the `this` parameter may 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 nonstandard `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, as follows:

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

- 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 a value of 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
  ```

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

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

  A nonstandard pointer-to-member declaration that appears outside a class declaration, such as the declaration of `T`, is normally invalid and would cause an error to be issued. However, for declarations that appear within a class

declaration, such as `A::T3`, this feature changes the meaning of a valid declaration. 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
```

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.

Anachronisms Accepted

Warning: Although the following anachronisms are accepted, it is not recommended that they be used. Unpredictable results can occur.

The following anachronisms to the Standard 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. Example:

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

Index

D