Cray C/C++ Reference Manual

004–2179–003

© 1996-1999 Silicon Graphics, Inc. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of Silicon Graphics, Inc.

LIMITED AND RESTRICTED RIGHTS LEGEND

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 Unpublished rights reserved under the Copyright Laws of the United States.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

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

Silicon Graphics is a registered trademark and the Silicon Graphics logo is a trademark of Silicon Graphics, Inc.

TotalView is a trademark of Bolt Beranek and Newman Inc. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited. VAX is a trademark of Digital Equipment Corporation.

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

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

New Features

Cray C/C++ Reference Manual 004–2179–003

This document supports the 6.3 release of the Cray C compiler and the 3.3 release of the Cray C++ compiler, which are contained within the 3.3 release of the Programming Environment.

Information on the Multi-Streaming Processor (MSP), which will be an optional feature, has been included in the 3.3 release of the Cray C/C++ Reference Manual, although it will not be implemented in the Cray C or C++ compiler until a revision release of the 3.3 Programming Environment. The following new information in this document relates to MSP:

- `-h stream`*n* — See Section 2.9.1, page 14
- `#pragma nostream` directive (Cray SV1 systems) — See Section 4.9.1, page 80
- `#pragma preferstream` directive (Cray SV1 systems) — See Section 4.9.2, page 80
- `m` argument to the `-h report` option — See Section 2.18.3, page 28

Record of Revision

Version	***Description***
2.0	January 1996 Original Printing. This manual supports the C and C++ compilers contained in the Cray C++ Programming Environment release 2.0. On all Cray Research systems, the C++ compiler is Cray C++ 2.0. On Cray Research systems with IEEE floating-point hardware, the C compiler is Cray Standard C 5.0. On Cray Research systems without IEEE floating-point hardware, the C compiler is Cray Standard C 4.0.
3.0	May 1997 This rewrite supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.0, which is supported on all systems except the Cray T3D system. On all supported Cray Research systems, the C++ compiler is Cray C++ 3.0 and the C compiler is Cray C 6.0.
3.0.2	March 1998 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.0.2, which is supported on all systems except the Cray T3D system. On all supported Cray Research systems, the C++ compiler is Cray C++ 3.0.2 and the C compiler is Cray C 6.0.2.
3.1	August 1998 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.1, which is supported on all systems except the Cray T3D system. On all supported Cray Research systems, the C++ compiler is Cray C++ 3.1 and the C compiler is Cray C 6.1.
3.2	January 1999 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.2, which is supported on all systems except the Cray T3D system. On all supported Cray Research systems, the C++ compiler is Cray C++ 3.2 and the C compiler is Cray C 6.2.
3.3	July 1999 This revision supports the C and C++ compilers contained in the Cray C++ Programming Environment release 3.3, which is supported on the Cray SV1, Cray C90, Cray J90, and Cray T90 systems running UNICOS 10.0.0.5 and later, and Cray T3E systems running UNICOS/mk 2.0.4 and later. On all supported Cray Research systems, the C++ compiler is Cray C++ 3.3 and the C compiler is Cray C 6.3.

Contents

Page

About This Manual **xv**

Related Publications . . . xv

Obtaining Publications . . . xvi

Conventions . . . xvi

Reader Comments . . . xviii

Introduction [1] **1**

Setting up the C/C++ Programming Environment . . . 1

General Compiler Description . . . 1

Cray C++ Compiler . . . 2

Cray Standard C Compiler . . . 2

Standard Template Library (STL) . . . 2

Compiler Commands [2] **3**

CC Command . . . 4

cc Command . . . 4

c89 Command . . . 4

cpp Command . . . 5

Command-line Options . . . 5

Language (Standard Conformance) . . . 6

-h [no]conform (CC, cc Commands), -h [no]stdc (cc Command) . . . 6

-h cfront (CC Command) . . . 7

-h [no]exceptions (CC Command) . . . 7

-h [no]anachronisms (CC Command) . . . 7

-h new_for_init (CC Command) . . . 7

-h [no]tolerant (cc Command) . . . 8

Page

Language (Templates) 8
-h [no]autoinstantiate (CC Command) 8
-h instantiate=*mode* (CC Command) 8
-h [no]implicitinclude (CC Command) 9
-h remove_instantiation_flags (CC Command) 9
-h prelink_local_copy (CC Command) 9
-h prelink_copy_if_nonlocal (CC Command) 9
Language (Virtual Functions) 9
-h forcevtbl, -h suppressvtbl (CC Command) 9
Language (General) 10
-h keep=*file* (CC Command) 10
-h restrict=*args* (CC, cc Commands) 10
-h [no]calchars (CC, cc Commands) 11
-h [no]signedshifts (CC, cc Commands) 11
-h [no]stack (CC, cc Commands) 12
Optimization (General) 12
-O [*level*] (CC, cc, c89 Commands) 12
-h [no]aggress (CC, cc Commands) 13
-h [no]intrinsics (CC, cc Commands) 13
-h [no]pattern (CC, cc Commands) 13
-h [no]overindex (CC, cc Commands) 13
Optimization (Multi-Streaming Processor) (Deferred Implementation) 14
-h stream*n* (CC, cc Commands) 14
Optimization (Vector) 14
-h [no]ivdep (CC, cc Commands) 15
-h vector*n* (CC, cc Commands) 15
-h [no]vsearch (CC, cc Commands) 16
Optimization (Task) 16

Page

-h task*n* (CC, cc Commands) 16
-h taskprivate (cc Command) 17
-h taskcommon, -h common (CC, cc commands) 17
-h [no]taskinner (CC, cc Commands) 17
-h [no]threshold (CC, cc Commands) 18
Optimization (Inline) 18
-h inline*n* (CC, cc Commands) 18
-h inlinefrom=*file* (CC, cc Commands) 19
Optimization (Scalar) 19
-h [no]interchange (CC, cc Commands) 19
-h scalar*n* (CC, cc Commands) 19
-h [no]align (CC, cc Commands) 20
-h [no]bl (CC, cc Commands) 20
-h [no]reduction (CC, cc Commands) 21
-h [no]zeroinc (CC, cc Commands) 21
Optimization (UNICOS/mk Specific) 21
-h pipeline*n* (CC, cc Commands) 21
-h [no]unroll (CC, cc Commands) 22
-h [no]jump (CC, cc Commands) 22
-h [no]split (CC, cc Commands) 22
Math 23
-h matherror=*method* (CC, cc Commands) 23
-h [no]fastmd (CC, cc Commands) 23
-h [no]fastmodulus (CC, cc Commands) 23
-h [no]ieeeconform (CC, cc Commands) 24
-h [no]fastfpdivide (CC, cc Commands) 24
-h [no]rounddiv (CC, cc Commands) 24
-h [no]trunc[=*n*] (CC, cc Commands) 25

	Page
Analysis Tools	25
-F (CC, cc Commands)	25
-h [no]atexpert (CC, cc Commands)	26
-h [no]apprentice (CC, cc Commands)	26
-h [no]listing (CC, cc Commands)	26
Debugging	26
-G *level* (CC, cc Commands) and -g (CC, cc, c89 Commands)	26
-h [no]bounds (cc Command)	27
-h indef, -h zero (CC, cc Commands)	27
Messages	28
-h msglevel_*n* (CC, cc Commands)	28
-h [no]message=*n*[: *n*...] (CC, cc Commands)	28
-h report=*args* (CC, cc Commands)	28
-h [no]abort (CC, cc Commands)	29
-h errorlimit[=*n*] (CC, cc Commands)	29
Compilation Phases	29
-E (CC, cc, c89, cpp Commands)	30
-P (CC, cc, cpp Commands)	30
-h feonly (CC, cc Commands)	30
-S (CC, cc Commands)	30
-c (CC, cc, c89 Commands)	30
-#, -##, and -### (CC, cc, cpp Commands)	31
-W*phase*["*opt*..."] (CC, cc Commands)	31
-Y*phase*,*dirname* (CC, cc, c89, cpp Commands)	32
Preprocessing	32
-I *incldir* (CC, cc, c89, cpp Commands)	32
-D *macro*[=*def*] (CC, cc, c89, cpp Commands)	33
-U *macro* (CC, cc, c89, cpp Commands)	34

Page

-M (CC, cc, cpp Commands) 34
-N (cpp Command) 34
-C (CC, cc, cpp Commands) 34
-h [no]pragma=*name*[: *name*...] (CC, cc Commands) 34
Loader 35
-l *libfile* (CC, cc, c89 Commands) 35
-L *libdir* (CC, cc, c89 Commands) 36
-o *outfile* (CC, cc, c89 Commands) 37
-d *string* (CC, cc Commands) 37
-s (CC, cc, c89 Commands) 37
General 37
-V (CC, cc, cpp Commands) 37
-X *npes* (CC, cc Commands) 38
-h ident=*name* (CC, cc Commands) 39
Command-line Examples 39
Environment Variables 41

Cray C and C++ Extensions [3] **43**

Restricted Pointers 43
Function Parameters 44
File Scope 45
Block Scope 45
Unrestricted Pointers 46
Comparison with #pragma ivdep 46
Implementation Limits 46
long long and unsigned long long Data Types 47
// Comments 48
Complex Data Types 48
Complex Usage 49

Page

Conversion to and from Complex 49
Arithmetic Conversion for Complex 50
Variable Length Arrays 50
Declarator Restrictions 50
Variable Length Array (VLA) Declarators 51
Function Declarators and Variable Length Arrays 52
Variable Length Array Type Definitions 53
sizeof Operator and Variable Length Arrays 54
goto Statements 55
switch Statement 55
setjmp and longjmp Functions 56
fortran Keyword 57
Hexadecimal Floating-point Constants 58
Arithmetic Conversions 59

#pragma Directives [4] 61
Protecting Directives 62
Directives in Cray C++ 62
Loop Directives 62
Alternative Directive Form: _Pragma 63
General Directives 63
besu Directive 63
[no]bounds Directive (C Compiler) 64
duplicate Directive (C Compiler) 65
message Directive 66
[no]opt Directive 66
uses_eregs Directive 66
soft Directive 67

Page

vfunction Directive 68
ident Directive 68
Instantiation Directives 68
Vectorization Directives 69
ivdep Directive 69
novector Directive 70
novsearch Directive 70
prefervector Directive 70
shortloop and shortloop128 Directives 71
Tasking Directives 71
parallel and endparallel Directives 72
taskloop Directive 72
endloop Directive 73
case and endcase Directives 73
guard and endguard Directives 74
taskprivate Directive (C Compiler) 74
taskshared Directive (C Compiler) 75
taskcommon Directive 76
common Directive 77
prefertask Directive 77
Arguments to Tasking Directives 78
Context Arguments 78
Work Distribution Arguments 78
Miscellaneous Arguments 79
Multi-Streaming Processor (MSP) Directives (Deferred Implementation) 79
#pragma nostream Directive 80
#pragma preferstream Directive 80
Scalar Directives 81
align Directive 81

Page

cache_align Directive . . . 81
cache_bypass Directive . . . 82
concurrent Directive . . . 83
nointerchange Directive . . . 83
noreduction Directive . . . 84
split Directive . . . 84
suppress Directive . . . 85
symmetric Directive . . . 86
unroll Directive . . . 87
Inlining Directives . . . 88
inline Directive . . . 89
noinline Directive . . . 89

Template Instantiation [5] **91**
Automatic Instantiation . . . 92
Instantiation Modes . . . 94
Instantiation #pragma Directives . . . 95
Implicit Inclusion . . . 97

Predefined Macros [6] **99**
Macros Required by the C and C++ Standards . . . 99
Macros Based on the Host Machine . . . 100
Macros Based on the Target Machine . . . 100
Macros Based on the Compiler . . . 101

Debugging C/C++ Code [7] **103**
Cray TotalView Debugger . . . 103
Compiler Debugging Options . . . 104

Interlanguage Communication [8] **107**
Interlanguage Communication with Cray Standard C and Cray C++ . . . 107

Page

Calling Assembly Language Functions from a C or C++ Function 107
Cray Assembly Language (CAL) Functions on UNICOS Systems 108
Cray Assembler for MPP (CAM) Functions on UNICOS/mk Systems 108
Calling Fortran Functions and Subroutines from a C or C++ Function 108
Requirements 108
Argument Passing 109
Array Storage 109
Logical and Character Data 111
Accessing Named Common from C/C++ 112
Accessing Blank Common from C/C++ 114
C and Fortran Example 116
Calling a Fortran Program from a C++ Program 119
Calling a C/C++ Function from an Assembly Language or Fortran Program 120
Calls between C and C++ Functions 123

Implementation-defined Behavior [9] **127**

Implementation-defined Behavior 127
Messages 127
Environment 127
Identifiers 128
Types 128
Characters 130
Wide Characters 131
Integers 131
Floating-point Arithmetic 132
Arrays and Pointers 135
Registers 136
Classes, Structures, Unions, Enumerations, and Bit Fields 136
Qualifiers 137

Page

Declarators . . . 137
Statements . . . 137
Exceptions . . . 138
System Function Calls . . . 138
Preprocessing . . . 138

Appendix A Libraries and Loaders 139

UNICOS Standard C and C++ Libraries . . . 139
UNICOS Loaders . . . 139
Loader for UNICOS Systems (SEGLDR) . . . 140
Loader for UNICOS/mk Systems (`cld`(1)) . . . 140

Appendix B Cray C/C++ Dialects 141

C++ Conformance . . . 141
Supported Features . . . 141
Unsupported Features . . . 144
C++ Anachronisms Accepted . . . 145
Extensions Accepted in Normal C++ Mode . . . 146
Extensions Accepted in C or C++ Mode . . . 147
C++ Extensions Accepted in `cfront` Compatibility Mode . . . 149

Appendix C Compiler Messages 157

Expanding Messages with the `explain` Command . . . 157
Controlling the Use of Messages . . . 158
Command-line Options . . . 158
Environment Options for Messages . . . 159
`ORIG_CMD_NAME` Environment Variable . . . 159
Message Severity . . . 160
Common System Messages . . . 161

Page

Appendix D Intrinsic Functions **163**

Index **167**

Figures

Figure 1. Character pointer format 136

Tables

Table 1. `-h` option descriptions 12

Table 2. `-h pragma` directive processing 35

Table 3. Cray Research systems data type mapping 129

Table 4. Summary of C/C++ intrinsic functions 164

About This Manual

This publication documents the commands, directives, language extensions, and other details specific to the Cray Research implementation of the Standard C and C++ languages on the following systems:

- Cray C90, Cray J90, Cray SV1, Cray T90, and Cray T90 IEEE systems running UNICOS 10.0.0.5 or later
- Cray T3E systems running UNICOS/mk 2.0.4 or later

It is assumed that readers of this manual have a working knowledge of the C and C++ programming languages.

Related Publications

The following documents contain additional information that may be helpful:

- *UNICOS User Commands Reference Manual*
- *UNICOS System Libraries Reference Manual*
- *Introducing CrayLibs*
- *Intrinsic Procedures Reference Manual*
- *Scientific Library Reference Manual*
- *Scientific Libraries User's Guide*
- *C++ Language System Release 3.0.1 Library Manual*, publication S3–2131
- *Cray C/C++ Ready Reference*
- *C++ Installation Guide*
- *Application Programmer's Library Reference Manual*
- *Application Programmer's I/O Guide*
- *Introducing the Program Browser*
- *Introducing the Cray TotalView Debugger*
- *Compiler Information File (CIF) Reference Manual*
- *Guide to Parallel Vector Applications*

- *Introducing the MPP Apprentice Tool*
- *Optimizing Application Code on UNICOS Systems*
- *Cray C++ Tools Library Reference Manual*, Rogue Wave document, *Tools.h++ Introduction and Reference Manual*, publication TPD-0005
- *Cray C++ Mathpack Class Library Reference Manual* by Thomas Keefer and Allan Vermeulen, publication TPD-0006
- *LAPACK.h++ Introduction and Reference Manual, Version 1*, by Allan Vermeulen, publication TPD-0010
- *Standard Template Library Programmer's Guide*, Silicon Graphics WWW site:

  `http://www.sgi.com/Technology/STL`

Obtaining Publications

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

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

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

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

The *Standard Template Library Programmer's Guide* (STL documentation) is available to view or download at the following URL:

`http://www.sgi.com/Technology/STL`

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.

`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers:

1	User commands
1B	User commands ported from BSD
2	System calls
3	Library routines, macros, and opdefs
4	Devices (special files)
4P	Protocols
5	File formats
7	Miscellaneous topics
7D	DWB-related information
8	Administrator commands

Some internal routines (for example, the `_assign_asgcmd_info`() routine) do not have man pages associated with them.

variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

The following machine naming conventions may be used throughout this document:

Term	Definition
Cray PVP systems	All configurations of Cray parallel vector processing (PVP) systems.

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

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

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

Reader Comments

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

You can contact us in any of the following ways:

- Send e-mail to the following address:

  `techpubs@sgi.com`

- Send a fax to the attention of “Technical Publications” at: +1 650 932 0801.
- Use the Feedback option on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com`

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

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

  For UNICOS or UNICOS/mk based operating systems or Cray Origin 2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

Technical Publications
SGI
1600 Amphitheatre Pkwy.
Mountain View, California 94043–1351

We value your comments and will respond to them promptly.

Introduction [1]

The Cray C++ Programming Environment contains both the Cray Standard C and the Cray C++ compilers. The Cray Standard C compiler conforms with the International Standards Organization (ISO) standard ISO/IEC 9899:1990 and the American National Standard Institute (ANSI) X3.159-1989. The Cray C++ compiler conforms with the ISO/ANSI Draft Proposed International Standard - Programming Language C++ document numbers X3J16/94-0158 and WG21/n0545.

The C and C++ compilers run on the following systems:

- Cray SV1, Cray C90, Cray J90, and Cray T90 systems running UNICOS 10.0.0.5 or later.
- Cray T3E systems running UNICOS/mk 2.0.4 or later.

Note: Throughout this manual, the differences between the Cray Standard C and the Cray C++ compilers are noted when appropriate. When there is no difference, the phrases *the compiler* or *the C/C++ compiler* refer to both compilers.

1.1 Setting up the C/C++ Programming Environment

The installation and configuration of the C/C++ programming environment uses a utility called `modules`, which is provided and installed as part of the release package. The `/opt/ctl/doc/README` file was distributed in the release package. It contains information on initializing the `module` command and initializing the environment.

The default programming environment is available to you after you have entered the following command:

```
module load modules PrgEnv
```

If you have questions on setting up the programming environment, contact your system support staff.

1.2 General Compiler Description

Because both the Cray Standard C and Cray C++ compilers are contained within the same programming environment, programmers compiling code written in C

should use the cc(1) or c89(1) commands to compile their source files, and programmers compiling code written in C++ should use the CC(1) command.

1.2.1 Cray C++ Compiler

The Cray C++ compiler consists of a preprocessor, a language parser, a prelinker, an optimizer, and a code generator. The Cray C++ compiler is invoked by a command called CC(1) in this manual, but may be renamed at individual sites. The CC(1) command is described in Section 2.1, page 4, and on the CC(1) man page. Command-line examples are shown in Section 2.23, page 39.

1.2.2 Cray Standard C Compiler

The Cray Standard C compiler consists of a preprocessor, a language parser, an optimizer, and a code generator. The Cray Standard C compiler is invoked by a command called cc or c89 in this manual, but may be renamed at individual sites. The cc and c89 commands are described in Section 2.1, page 4, and on the CC man page. Command-line examples are shown in Section 2.23, page 39.

1.3 Standard Template Library (STL)

The Standard Template Library (STL) is a C++ library of container classes, algorithms, and iterators; it provides many of the basic algorithms and data structures of computer science. The STL is a generic library, meaning that its components are heavily parameterized: almost every component in the STL is a template. You should make sure that you understand how templates work in C++ before you use the STL. The *Standard Template Library Programmer's Guide* is available online only with the Programming Environment 3.1 release. You can also view the *Standard Template Library Programmer's Guide* from the following World Wide Web site:

`http://www.sgi.com/Technology/STL/index.html.`

Compiler Commands [2]

This section describes the following commands and the environment variables necessary to execute the compilers associated with the Cray C++ programming environment:

- CC command invokes the Cray C++ compiler.
- cc command invokes the Cray Standard C compiler.
- c89 command invokes the Cray Standard C compiler. This command is a subset of the cc command and conforms with the POSIX standard (P1003.2, Draft 12).
- cpp command explicitly invokes the preprocessor component of the Cray Standard C compiler.

The compilation process, if successful, creates an absolute binary file, named a.out by default, that reflects the contents of the source code and any referenced library functions. This binary file, a.out, can then be executed on the target system. For example, the following sequence compiles file mysource.c and executes the resulting executable program:

```
cc mysource.c
./a.out
```

With the use of appropriate options, compilation can be terminated to produce one of several intermediate translations, including relocatable object files (-c option), assembly source expansions (-S option), or the output of the preprocessor phase of the compiler (-P or -E option). In general, the intermediate files can be saved and later resubmitted to the CC, cc, or c89 commands, with other files or libraries included as necessary.

By default, the CC, cc, and c89 commands automatically call the loader, which creates an executable file. If only one source file is specified, the object file is deleted. If more than one source file is specified, the object files are retained. The following example creates object files ***file1***.o, ***file2***.o, and ***file3***.o, and the executable file a.out:

```
CC file1.c file2.c file3.c
```

The following command creates the executable file a.out only:

```
CC file.c
```

2.1 CC Command

The CC command invokes the Cray C++ compiler. The CC command accepts C++ source files that have the .c, .C, .i, .c++, .C++, .cc, .cxx, .Cxx, .CXX, .CC, and .cpp suffixes; object files with the .o suffix; library files with the .a suffix; and assembler source files with the .s suffix.

The CC command format is as follows:

```
CC [-c] [-C] [-d string] [-D macro[=def]] [-E] [-F] [-g]
   [-G level] [-h arg] [-I incldir] [-l libfile] [-L libdir] [-M]
   [-o outfile] [-O [level]] [-P] [-s] [-S] [-U macro] [-V]
   [-Wphase,["opt..."]] [-Xnpes] [-Yphase,dirname] [-#] [-##]
   [-###]  files...
```

2.2 cc Command

The cc command invokes the Cray Standard C compiler. The cc command accepts C source files that have the .c and .i suffixes; object files with the .o suffix; library files with the .a suffix; and assembler source files with the .s suffix.

The cc command format is as follows:

```
cc [-c] [-C] [-d string] [-D macro[=def]] [-E] [-F] [-g]
   [-G level] [-h arg] [-I incldir] [-l libfile] [-L libdir] [-M]
   [-o outfile] [-O [level]] [-P] [-s] [-S] [-U macro] [-V]
   [-Wphase,["opt..."]] [-X npes] [-Yphase,dirname] [-#] [-##]
   [-###]  files...
```

2.2.1 c89 Command

The c89 command invokes the Cray Standard C compiler. This command is a subset of the cc command and conforms with the POSIX standard (P1003.2, Draft 12). The c89 command accepts C source files that have the .c and .i suffixes; object files with the .o suffix; library files with the .a suffix; and assembler source files with the .s suffix.

The c89 command format is as follows:

```
c89 [-c] [-D macro[=def]] [-E] [-g] [-I incldir] [-l libfile]
   [-L libdir] [-o outfile] [-O [level]] [-s] [-U macro]
   [-Yp,dirname] files...
```

2.2.2 cpp Command

The cpp command explicitly invokes the preprocessor component of the Cray Standard C compiler. Most cpp options are also available from the CC, cc, and c89 commands.

The cpp command format is as follows:

```
cpp [-C] [-D macro[=def]] [-E] [-I incldir] [-M] [-N] [-P]
   [-U macro] [-V] [-Yphase,dirname] [-#] [-##] [-###]
   [infile[outfile]]
```

The *infile* and *outfile* files are, respectively, the input and output for the preprocessor. If you do not specify these arguments, input is defaulted to standard input (stdin) and output to standard output (stdout). Specifying - for *infile* also indicates standard input.

2.3 Command-line Options

This following subsections describe options for the CC, cc, c89, and cpp commands, which are grouped by the following functions:

Language

Optimization

Math

Analysis tools

Debugging

Messages

Compilation phases

Preprocessing

Loader

General

Options other than those described here are passed to the loader; see the appropriate man page for loader options.

At the beginning of each subsection, information is included about exceptions to the default option, the systems that use the option, the type of floating-point representation used (IEEE and/or Cray), and the commands that will accept the option. Unless otherwise noted, the following default information applies to each option:

Default option:	None
Operating System:	UNICOS and UNICOS/mk
Floating-point:	IEEE and Cray

If conflicting options are specified, the option specified last on the command line overrides the previously specified option. Exceptions to this rule are noted in the individual descriptions of the options.

Most `#pragma` directives override corresponding command-line options. For example, `#pragma _CRI novsearch` overrides the `-h vsearch` option. `#pragma _CRI novsearch` also overrides the `-h vsearch` option implied by the `-h vector2` or `-O2` option. Exceptions to this rule are noted in descriptions of options or `#pragma` directives.

2.4 Language (Standard Conformance)

The following sections describe standard conformance language options.

2.4.1 -h [no]conform (CC, cc Commands), -h [no]stdc (cc Command)

Default option: `-h noconform`, `-h nostdc`

The `-h conform` and `-h stdc` options specify strict conformance to the ISO C standard or the draft ISO C++ standard. The `-h noconform` and `-h nostdc` options specify partial conformance to the standard. `-h exceptions` is enabled by `-h conform` in C++.

Note: The `c89` command does not accept the `-h conform` or `-h stdc` option. It is enabled by default when the command is issued.

2.4.2 -h cfront (CC Command)

The -h cfront option causes the C++ compiler to accept or reject constructs that were accepted by previous cfront-based compilers (such as Cray C++ 1.0), but which are not accepted in the ANSI/ISO draft standard. The -h anachronisms option is implied when -h cfront is specified.

2.4.3 -h [no]exceptions (CC Command)

Default option: -h noexceptions

The -h exceptions option enables support for exception handling. The -h noexceptions option issues an error whenever an exception construct, a try block, a throw expression, or a throw specification on a function declaration is encountered. -h exceptions is enabled by -h conform.

2.4.4 -h [no]anachronisms (CC Command)

Default option: -h noanachronisms

The -h [no]anachronisms option enables or disables anachronisms in Cray C++. This option is overridden by -h conform.

2.4.5 -h new_for_init (CC Command)

The -h new_for_init option enables the new scoping rules for a declaration in a for-init-statement. This means that the new (standard-conforming) rules are in effect, which means that the entire for statement is *wrapped* in its own implicitly generated scope. -h new_for_init is implied by the -h conform option.

The following is the result of the scoping rule:

```
{
  .
  .
  .
  for (int i = 0; i < n; i++)  {
    .
    .
    .
  } // scope of i ends here for -h new_for_init
  .
  .
```

```
    .
}  // scope of i ends here by default
```

2.4.6 -h [no]tolerant (cc Command)

Default option: -h notolerant

The -h tolerant option allows older, less standard C constructs to facilitate porting of code written for previous C compilers. Errors involving comparisons or assignments of pointers and integers become warnings. The compiler generates casts so that the types agree. With -h notolerant, the compiler is intolerant of the older constructs.

This option can be specified on the same line with -O3 or any combination of -h scalar3, -h vector3, or -h task3. The combination of -h tolerant with these options causes the compiler to tolerate accessing an object with one type through a pointer to an entirely different type. For example, a pointer to int might be used to access an object declared with type double. Such references violate the C standard and should be eliminated if possible. They can reduce the effectiveness of alias analysis and inhibit optimization.

2.5 Language (Templates)

The following sections describe template language options. See Chapter 5, page 91 for more information on template instantiation.

2.5.1 -h [no]autoinstantiate (CC Command)

Default option: -h autoinstantiate

The -h [no]autoinstantiate option enables or disables automatic instantiation of templates by the Cray C++ compiler.

2.5.2 -h instantiate=*mode* (CC Command)

Default option: -h instantiate=none

Normally, during compilation of a source file, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by using the -h instantiate=*mode* option. *mode* is specified as none (the default), used, all, or local.

2.5.3 -h [no]implicitinclude (CC Command)

Default option: -h implicitinclude

The -h [no]implicitinclude option enables or disables implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

2.5.4 -h remove_instantiation_flags (CC Command)

The -h remove_instantiation_flags option causes the prelinker to recompile all the sources to remove all instantiation flags.

2.5.5 -h prelink_local_copy (CC Command)

The -h prelink_local_copy indicates that only local files (for example, files in the current directory) are candidates for assignment of instantiations.

2.5.6 -h prelink_copy_if_nonlocal (CC Command)

The -h prelink_copy_if_nonlocal option specifies that assignment of an instantiation to a nonlocal object file will result in the object file being recompiled in the current directory.

2.6 Language (Virtual Functions)

The following sections describe virtual function options.

2.6.1 -h forcevtbl, -h suppressvtbl (CC Command)

The -h forcevtbl option forces the definition of virtual function tables in cases where the heuristic methods used by the compiler to decide on definition of virtual function tables provide no guidance. The -h suppressvtbl option suppresses the definition of virtual function tables in these cases.

The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first noninline, nonpure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity).

The -h forcevtbl option differs from the default behavior in that it does not force the definition to be local.

2.7 Language (General)

The following sections describe general language options.

2.7.1 `-h keep=`*file* (`CC` Command)

When the `-h keep=`*file* option is specified, the static constructor/destructor object (`.o`) file is retained as *file*. This option is useful when linking `.o` files on a system that does not have a C++ compiler. The use of this option requires that the `main` function must be compiled by C++ and the static constructor/destructor function must be included in the link. With these precautions, mixed object files (files with `.o` suffixes) from C and C++ compilations can be linked into executables by using the loader command instead of the `CC` command.

2.7.2 `-h restrict=`*args* (`CC`, `cc` Commands)

The `-h restrict=`*args* option globally instructs the compiler to treat certain classes of pointers as restricted pointers. You can use this option to enhance optimizations. This includes vectorization on Cray PVP systems.

Classes of affected pointers are determined by the value contained in *args*, as follows:

args	Description
`a`	All pointers to object and incomplete types are to be considered restricted pointers, regardless of where they appear in the source code. This includes pointers in `class`, `struct`, and `union` declarations, type casts, function prototypes, and so on.
`f`	All function parameters that are pointers to objects or incomplete types can be treated as restricted pointers.
`t`	All `this` parameters can be treated as restricted pointers (C++ only).

The *args* arguments instruct the compiler to assume that, in the current compilation unit, each pointer (`=a`), or each pointer that is a function parameter (`=f`), or each `this` pointer (`=t`) points to a unique object. This assumption eliminates those pointers as sources of potential aliasing, and may allow additional vectorization or other optimizations. These options cause only data dependencies from pointer aliasing to be ignored, rather than all data

dependencies, so they can be used safely for more programs than the `-h ivdep` option.

Caution: Like `-h ivdep`, the arguments make assertions about your program that, if incorrect, can introduce undefined behavior. You should not use `-h restrict=a` if, during the execution of any function, an object is modified and that object is referenced through either of the following:

- Two different pointers
- The declared name of the object and a pointer

The `-h restrict=f` and `-h restrict=t` options are subject to the analogous restriction, with "function parameter pointer" replacing "pointer."

2.7.3 -h [no]calchars (CC, cc Commands)

Default option: `-h nocalchars`

The `-h calchars` option allows the use of the `@` and `$` characters in identifier names. This option is useful for porting codes in which identifiers include these characters. With `-h nocalchars`, these characters are not allowed in identifier names.

Caution: Use this option with extreme care, because identifiers with these characters are within Cray Research name space and are included in many Cray Research library identifiers, internal compiler labels, objects, and functions. You must prevent conflicts between any of these uses, current or future, and identifier declarations or references in your code; any such conflict is an error.

2.7.4 -h [no]signedshifts (CC, cc Commands)

Default option: `-h signedshifts`

(UNICOS/mk systems) The `-h [no]signedshifts` option affects the result of the right shift operator. For the expression `e1 >> e2` where `e1` has a signed type, when `-h signedshifts` is in effect, the vacated bits are filled with the sign bit of `e1`. When `-h nosignedshifts` is in effect, the vacated bits are filled with zeros, identical to the behavior when `e1` has an unsigned type. The `-h nosignedshifts` option forces the `>>` operator to have the same behavior on UNICOS/mk operating systems as on UNICOS operating systems.

2.7.5 -h [no]stack (CC, cc Commands)

Default option: -h stack

(UNICOS systems) The -h stack option specifies the stack calling sequence. The -h nostack option specifies the standard UNICOS static calling sequence and prevents the use of reentrant or recursive functions. The -h nostack option disables autotasking, all tasking directives, and the -h taskcommon and -h taskprivate options.

2.8 Optimization (General)

The following sections describe general optimization options.

2.8.1 -O [*level*] (CC, cc, c89 Commands)

Default option: Equivalent to the appropriate -h option

The -O *level* option specifies the optimization level for a group of compiler features. Specifying -O with no arguments is the same as not specifying the -O option. A value of 0, 1, 2, or 3 sets that level of optimization for the -h inline, -h scalar, -h task, and -h vector options.

For example, -O2 is equivalent to the following:

```
-h inline2,scalar2,task2,vector2
```

Optimization features specified by -O are equivalent to the following -h options (task*n* is ignored on UNICOS/mk systems):

Table 1. -h option descriptions

-h option	Description location
-h inline*n*	Section 2.12.1, page 18
-h scalar*n*	Section 2.13.2, page 19
-h task*n*	Section 2.11.1, page 16
-h vector*n*	Section 2.10.2, page 15

2.8.2 -h [no]aggress (CC, cc Commands)

Default option: -h noaggress

The -h aggress option provides greater opportunity to optimize loops that would otherwise by inhibited from optimization due to an internal compiler size limitation. -h noaggress leaves this size limitation in effect.

With -h aggress, internal compiler tables are expanded to accommodate larger loop bodies. This option can increase the compilation's time and memory size. On UNICOS systems, this option also disables the limit on the number of vector updates in a single loop. On UNICOS/mk systems, this option enables the compiler to aggressively assign registers and schedule instructions.

2.8.3 -h [no]intrinsics (CC, cc Commands)

Default option: -h intrinsics

The -h intrinsics option allows the use of intrinsic hardware functions, which allow direct access to some Cray Research hardware instructions or generate inline code for some functions. This option has no effect on specially-handled library functions.

Intrinsic functions are described in Appendix D, page 163.

2.8.4 -h [no]pattern (CC, cc Commands)

Default option: -h pattern

The -h [no]pattern option globally enables or disables pattern matching. Pattern matching is on by default, but takes effect only when -h vector2 or -h scalar2 or greater are specified.

2.8.5 -h [no]overindex (CC, cc Commands)

Default option: -h nooverindex

The -h overindex option declares that there are array subscripts that index a dimension of an array that is outside the declared bounds of that array. The -h nooverindex option declares that there are no array subscripts that index a dimension of an array that is outside the declared bounds of that array.

2.9 Optimization (Multi-Streaming Processor) (Deferred Implementation)

The following sections describe Multi-Streaming Processor (MSP) options. For information on MSP `#pragma` directives, see Section 4.9, page 79.

> **Note:** MSP is an optional feature. To determine whether MSP is enabled on your system, enter the `sysconf` command at your system prompt. The `HARDWARE` output field contains the `NMSP=` field that shows the number of MSPs configured. For more information, see the `sysconf`(1) man page.

2.9.1 `-h stream`*n* (`CC`, `cc` Commands)

(Cray SV1 systems) The `-h stream`*n* option specifies the level of automatic multi-streaming processing (MSP) optimizations to be performed. Generally, vectorized applications that execute on a 1–processor system can expect to execute up to 4 times faster on a processor with multi-streaming enabled.

Argument *n* can be one of the following:

n	Description
0	No automatic multi-streaming optimizations are performed.
1	Conservative automatic multi-streaming optimizations. Automatic multi-streaming optimization is limited to inner vectorized loops. MSP operations performed generate the same results that would be obtained from scalar optimizations; for example, no floating-point reductions are performed.
2	Moderate automatic multi-streaming optimizations. Automatic multi-streaming optimization is performed on loop nests.
3	Aggressive automatic multi-streaming optimizations. Automatic multi-streaming optimization is performed as with `stream2`.

2.10 Optimization (Vector)

The following sections describe vector optimization options.

2.10.1 `-h [no]ivdep` (`CC`, `cc` Commands)

Default option: `-h noivdep`

The `-h ivdep` option instructs the compiler to ignore vector dependencies for all loops. This is useful for vectorizing loops that contain pointers. With `-h noivdep`, loop dependencies inhibit vectorization. To control loops individually, use the `#pragma ivdep` directive, as discussed in Section 4.7.1, page 69.

This option can also be used with "vectorization-like" optimizations on UNICOS/mk systems. See Section 4.7, page 69, for more information.

Caution: This option should be used with extreme caution because incorrect results can occur if there is a vector dependency within a loop. Combining this option with inlining is dangerous because inlining can introduce vector dependencies.

2.10.2 `-h vector`*n* (`CC`, `cc` Commands)

Default option: `-h vector2`

The `-h vector`*n* option specifies the level of automatic vectorizing to be performed. Vectorization results in dramatic performance improvements with a small increase in object code size. Vectorization directives are unaffected by this option.

On UNICOS/mk systems, the C/C++ compiler can do "vectorization-like" optimizations on loops that contain calls to certain functions. These optimizations are enabled/disabled with this option. See Section 4.7, page 69, for more information on these optimizations.

Argument *n* can be one of the following:

n	Description
`0`	No automatic vectorization.
`1`	Conservative automatic vectorization. On UNICOS systems, automatic vectorization is performed. Search loops and reduction loops are not vectorized.
`2`	Moderate automatic vectorization. On UNICOS systems, automatic vectorization is performed as with `vector1`, and vectorization of search loops and reduction loops is added.

3	Aggressive automatic vectorization. Automatic vectorization is performed as with `vector2` and restructuring of loops is done to improve vectorization. Also, the aliasing assumptions specified in the standard are used (for example, it is assumed that no aliasing will occur between two pointers to different structure types). On UNICOS/mk systems, "vectorization-like" optimizations are performed. See Section 4.7, page 69, for more information.

Vectorization directives are described in Section 4.7, page 69.

2.10.3 `-h [no]vsearch` (`CC`, `cc` Commands)

Default option: `-h vsearch`

(UNICOS systems) The `-h vsearch` option enables vectorization of all search loops. With `-h novsearch`, the default vectorization level applies. The `novsearch` directive is discussed in Section 4.7.3, page 70. This option is affected by the `-h vector`*n* option (see Section 2.10.2, page 15).

2.11 Optimization (Task)

The following sections describe task optimization options.

2.11.1 `-h task`*n* (`CC`, `cc` Commands)

Default option: `-h task0`

The `-h task`*n* option specifies the level of automatic tasking (Autotasking) to be performed. Tasking allows segments of code to execute in parallel on multiple processors. This option has no effect on tasking directives. Tasking and tasking directives are described in Section 4.8, page 71.

> **Note:** The `-h task`*n* option is accepted and ignored on UNICOS/mk systems.

Argument *n* can be one of the following:

n	Description
0	No Autotasking.
1	Conservative Autotasking. Same as `task0` in this release.
2	Moderate Autotasking. Same as `task0` in this release.

`3`	Aggressive Autotasking. This includes loop restructuring for improved tasking performance. Aliasing assumptions specified in the standard are also used; for example, it is assumed that no aliasing will occur between two pointers to different structure types.

Autotasking is disabled when `-h nostack` is specified.

2.11.2 `-h taskprivate` (`cc` Command)

This option gives task private status to all statically-allocated objects in the program. It has no effect if `-h nostack` is specified.

Unlike `-h taskcommon`, initialized objects can be made private to each task with the `-h taskprivate` option. They are initialized at startup for each task prior to the execution of the main entry point.

For information on tasking and tasking directives, see Section 4.8, page 71.

2.11.3 `-h taskcommon`, `-h common` (`CC`, `cc` commands)

Default option: `-h common`

The `-h taskcommon` option gives task common status to all statically-allocated objects in the program. It has no effect if `-h nostack` is specified. The `-h common` option gives `common` (as opposed to `taskcommon`) status to all global objects in the program. Tasking and tasking directives are described in Section 4.8, page 71.

Objects that are initialized will not be marked as task common. The `-h taskprivate` option can be used to make these objects private to each task and be initialized at startup for each task prior to the execution of the main entry point.

2.11.4 `-h [no]taskinner` (`CC`, `cc` Commands)

Default option: `-h notaskinner`

(UNICOS systems) Autotasking attempts to maximize the amount of parallel work in a taskable loop by interchanging the loop outwards. Sometimes this fails and a taskable loop remains innermost. By default, such a remaining innermost and taskable loop will not task if, at compile time, sufficient parallel work cannot be found. The `-h taskinner` option enables tasking of the innermost loop with a run-time threshold check to ensure that there is sufficient

parallel work in the loop. Aggressive Autotasking (`-h task3`) must also be specified for this option to take effect.

2.11.5 `-h [no]threshold` (`CC`, `cc` Commands)

Default option: `-h threshold`

(UNICOS systems) The `-h [no]threshold` option enables or disables generation of run-time threshold testing for autotasked loops. Aggressive Autotasking (`-h task3`) must also be specified for this option to take effect.

2.12 Optimization (Inline)

The following sections describe inline optimization options.

2.12.1 `-h inline`*n* (`CC`, `cc` Commands)

Default option: `-h inline1`

The `-h inline`*n* option specifies the level of inlining to be performed. Inlining eliminates the overhead of a function call and increases the opportunities for other optimizations. Inlining can also increase object code size. Inlining directives and the `inline` keyword are unaffected by this option.

Following are the values for the *n* argument:

n	Description
`0`	No inlining is performed.
`1`	Conservative inlining; performed only on functions implicitly marked by the `inline` keyword (C++) or a `#pragma _CRI inline` directive (C or C++) or on functions defined inside a class definition (C++).
`2`	Moderate automatic inlining; includes level `1` plus some automatic inlining.

3	Aggressive automatic inlining; all functions are candidates for inlining except those specifically marked with a `#pragma noinline` directive.

2.12.2 `-h inlinefrom=`*file* (`CC`, `cc` Commands)

The `-h inlinefrom=`*file* option specifies inline code expansion of all functions defined in *file* by the C++ compiler. For the *file* argument, enter the name of a file that contains one or more functions or enter the name of a directory that contains one or more source files.

Every function defined in the file or directory is inlined unless the call is within the scope of a `#pragma _CRI noinline` directive.

2.13 Optimization (Scalar)

The following sections describe scalar optimization options.

2.13.1 `-h [no]interchange` (`CC`, `cc` Commands)

Default option: `-h interchange`

The `-h interchange` option instructs the compiler to attempt to interchange all loops, a technique that is used to gain performance by having the compiler swap an inner loop with an outer loop. The compiler attempts the interchange only if the interchange will increase performance. Loop interchange is performed only at scalar optimization level 2 or higher.

The `-h nointerchange` option prevents the compiler from attempting to interchange any loops. To disable interchange of loops individually, use the `#pragma nointerchange` directive.

2.13.2 `-h scalar`*n* (`CC`, `cc` Commands)

Default option: `-h scalar1`

The `-h scalar`*n* option specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option (see Section 4.10, page 81).

Argument *n* can be one of the following:

n	Description
0	No automatic scalar optimization. The -h nobl, -h nofastfpdivide, -h nofastmd, -h nofastmodulus, -h matherror=errno, and -h zeroinc options are implied by -h scalar0.
1	Conservative automatic scalar optimization. This level implies -h fastmd, -h fastfpdivide, -h matherror=abort, and -h nozeroinc and causes automatic loop alignment to be performed.
2	Moderate automatic scalar optimization. The scalar optimizations specified by scalar1 are performed.
3	Aggressive automatic scalar optimization. The scalar optimizations specified by scalar2 are performed and -h fastmodulus and -h bl are implied.

2.13.3 -h [no]align (CC, cc Commands)

Default option: -h noalign

(UNICOS systems) The -h align option specifies that all functions defined in the file are to be automatically aligned on instruction buffer boundaries. This alignment can significantly improve performance for small, frequently called functions. With -h noalign, automatic function alignment is not done.

To control alignment of functions individually, use the align directive. For more information on the align directive and function alignment, see Section 4.10.1, page 81.

2.13.4 -h [no]bl (CC, cc Commands)

Default option: -h nobl

The -h bl option specifies a faster, but potentially unsafe, form of bottom loading. -h nobl dictates that this technique is not used. This option is affected by the scalar optimization level (see Section 2.13.2, page 19).

2.13.5 `-h [no]reduction` (`CC`, `cc` Commands)

Default option: `-h reduction`

On UNICOS systems, the `-h reduction` option instructs the compiler to enable vectorization of all reduction loops. On UNICOS/mk systems, the `-h reduction` option instructs the compiler to rewrite some multiplication operations to be a series of addition operations. With `-h noreduction`, these optimizations are not done. This option is affected by the `-h scalar`***n*** option (see Section 2.13.2, page 19). Reduction loops and the `noreduction` directive are discussed in Section 4.10.6, page 84.

2.13.6 `-h [no]zeroinc` (`CC`, `cc` Commands)

Default option: `-h nozeroinc`

The `-h nozeroinc` option improves run-time performance by causing the compiler to assume that constant increment variables (CIVs) in loops are not incremented by expressions with a value of 0.

The `-h zeroinc` option causes the compiler to assume that some CIVs in loops might be incremented by 0 for each pass through the loop, preventing generation of optimized code. For example, in a loop with index *i*, the expression *expr* in the statement *i* += *expr* can evaluate to 0. This rarely happens in actual code. `-h zeroinc` is the safer and slower option. This option is affected by the `-h scalar`***n*** option (see Section 2.13.2, page 19).

2.14 Optimization (UNICOS/mk Specific)

The following sections describe UNICOS/mk specific compiler options.

2.14.1 `-h pipeline`***n*** (`CC`, `cc` Commands)

Default option: `-h pipeline0`

(UNICOS/mk systems) The `-h pipeline`***n*** option specifies two levels of software pipelining; on and off. The following are the various software pipelining levels and their types of operations:

n	Description
0	No pipelining. Default.
1	Conservative pipelining. Only safe operator reassociations are performed. Numeric results obtained at this level do not differ from results obtained at level 0.
2	Moderate pipelining. Same as `pipeline1`.
3	Aggressive pipelining. Same as `pipeline1`.

2.14.2 `-h [no]unroll` (`CC`, `cc` Commands)

Default option: `-h nounroll`

(UNICOS/mk systems) The `-h unroll` option instructs the compiler to attempt to unroll all loops generated by the compiler. This technique is intended to increase single processor performance at the cost of increasing compile time and executable size. To control unrolling of loops individually, use the `unroll` directive. For more information on this directive and loop unrolling, see Section 4.10.10, page 87.

Note: On UNICOS systems, `-h unroll` is enabled at all times.

2.14.3 `-h [no]jump` (`CC`, `cc` Commands)

Default option: `-h jump`

(UNICOS/mk systems) The `-h jump` option generates jumps instead of branches to external functions. Branches provide slightly better performance. However, branches are limited in the distance to which they can transfer control; jumps have no such limitation. For large programs you may need to use `-h jump` with files that generate calls to functions loaded at too great a distance.

2.14.4 `-h [no]split` (`CC`, `cc` Commands)

Default option: `-h nosplit`

(UNICOS/mk systems) The `-h split` option instructs the compiler to attempt to split all loops generated by the compiler into sets of smaller loops. This technique is intended to increase single processor performance on UNICOS/mk systems by reducing thrashing of Cray T3E system hardware stream buffers. To control splitting of loops individually, use the `split` and `nosplit` directives.

For more information on these directives and on loop splitting, see Section 4.10.7, page 84.

2.15 Math

The following sections describe compiler options with regard to math functions.

2.15.1 `-h matherror=`*method* (`CC`, `cc` Commands)

Default option: `-h matherror=abort`

The `-h matherror=`*method* option specifies the method of error processing used if a standard math function encounters an error. The *method* argument can have one of the following values:

method	Description
`abort`	If an error is detected, `errno` is not set. Instead a message is issued and the program aborts. On Cray Research systems with IEEE floating-point hardware, an exception may be raised.
`errno`	If an error is detected, `errno` is set and the math function returns to the caller. This method is implied by the `-h conform`, `-h scalar0`, `-O0`, `-Gn`, and `-g` options.

2.15.2 `-h [no]fastmd` (`CC`, `cc` Commands)

Default option: `-h fastmd`

(UNICOS systems) The `-h fastmd` option generates shorter code sequences for `int` variables when doing multiply, divide, or comparison operations, or when converting to and from floating-point operations, but allows for only 46 bits of significance. With `-h nofastmd`, this action is disabled. This option is affected by the scalar optimization level (see Section 2.13.2, page 19).

2.15.3 `-h [no]fastmodulus` (`CC`, `cc` Commands)

Default option: `-h nofastmodulus`

(UNICOS systems) The `-h fastmodulus` option generates shorter code sequences for `int` variables used with the modulus operator (`%`), but allows only 46 significant bits. This option is affected by the scalar optimization level, (see Section 2.13.2, page 19).

2.15.4 -h [no]ieeeconform (CC, cc Commands)

Default option: -h noieeeconform

Floating-point: IEEE only

The -h ieeeconform option causes the resulting executable code to conform more closely to the IEEE floating-point standard (ANSI/IEEE Std 754–1985). Use of this option disables many arithmetic identity optimizations and may result in significantly slower code.

When -h noieeeconform is in effect, the compiler optimizes expressions such as x != x to 0 and x/x to 1 (where x has floating type). With the -h ieeeconform option in effect, these and other similar arithmetic identity optimizations are not performed. Optimizations on integral types are not affected by this option.

The -h ieeeconform option also turns on a scaled complex divide, which increases the range of complex values that can be handled without producing an underflow or an overflow.

The -h ieeeconform option overrides the -h fastfpdivide option.

2.15.5 -h [no]fastfpdivide (CC, cc Commands)

Default option: -h fastfpdivide

Floating-point: IEEE only

The -h fastfpdivide option decomposes a floating-point divide into a multiply-by-reciprocal in situations where a performance gain can be realized. For example, this option is useful in loops that contain divides with a loop-invariant divisor or sequences of divides with the same divisor. If the option is enabled, you could see slight numerical differences from compiles for which the option is not enabled. You could also see numerical differences between instances of the same computation for the same compile, depending on the context of the computation.

This option is affected by the scalar optimization level (see Section 2.8.1, page 12).

The -h ieeeconform option overrides the -h fastfpdivide option.

2.15.6 -h [no]rounddiv (CC, cc Commands)

Default option: -h norounddiv

Floating-point: Cray only

The `-h [no]rounddiv` option enables or disables strong rounding of all floating-point divide operations to allow the compiler to produce more symmetric results. Strong rounding ensures that all floating-point divide operations whose exact result is an integer will have an absolute value slightly greater than the integer value. This ensures that if these floating-point values are converted back into integer values they will represent the expected results. For example, results such as 2.99999..., when converted, will be treated as 3.

2.15.7 `-h [no]trunc[=n]` (CC, cc Commands)

Default option: `-h notrunc`

Floating-point: Cray only

The `-h trunc[=n]` option specifies truncation of the last *n* bits (range of *n*: $0 \leq n \leq 47$) of single-precision floating-point arithmetic. This option has no effect on double precision operations, function return values, and compile-time constants.

This option is useful for identifying numerically unstable algorithms. `-h trunc` with no argument is equivalent to `-h trunc=0`; that is, the assembler truncation instructions for floating-point arithmetic are generated, and the results from floating operations are truncated by `0` bits. The `-h notrunc` option generates assembler rounding instructions for floating arithmetic.

2.16 Analysis Tools

The following sections describe compiler options that support analysis tools.

2.16.1 `-F` (CC, cc Commands)

(UNICOS systems) The `-F` option enables the generation of additional run-time code that is needed by Flowtrace. Flowtrace is a program analysis tool that displays the call tree of a program and the amount of time spent in each function. See the `flowtrace`(7) man page for more information.

2.16.2 `-h [no]atexpert` (`CC`, `cc` Commands)

Default option: `-h noatexpert`

(UNICOS systems) The `-h atexpert` option generates additional run-time code needed to support the ATExpert tool (`atexpert`). With `-h noatexpert`, this code is not generated.

The ATExpert tool (`atexpert`) has been developed by Cray Research for accurately measuring and graphically displaying tasking performance from a job run on an arbitrarily loaded Cray Research system. For further information on using ATExpert to analyze your tasked programs, see the `atexpert`(7) man page.

2.16.3 `-h [no]apprentice` (`CC`, `cc` Commands)

Default option: `-h noapprentice`

(UNICOS/mk systems) For each source file compiled, the `-h apprentice` option enables or disables the creation of a compiler information file for use by the MPP Apprentice tool.

The MPP Apprentice tool helps you tune the performance of Cray T3E applications. It is documented in *Introducing the MPP Apprentice Tool*.

2.16.4 `-h [no]listing` (`CC`, `cc` Commands)

Default option: `-h nolisting`

(UNICOS systems) The `-h listing` option generates a pseudo assembly language listing. With `-h nolisting`, this listing is not produced. The listing file name is the same as the source file name, with the suffix replaced by a `.L`.

2.17 Debugging

The following sections describe compiler options used for debugging.

2.17.1 `-G` *level* (`CC`, `cc` Commands) and `-g` (`CC`, `cc`, `c89` Commands)

The `-g` and `-G` *level* options enable the generation of debugging information that is used by symbolic debuggers such as `totalview`. These options allow debugging with breakpoints. For the `-G` option, *level* indicates the following:

level	Optimization	Breakpoints allowed on
`f`	Full	Function entry and exit
`p`	Partial	Block boundaries
`n`	None	Every executable statement

More extensive debugging (such as full) permits greater optimization opportunities for the compiler. Debugging at any level may inhibit some optimization techniques, such as inlining.

The `-g` option is equivalent to `-Gn`. The `-g` option is included for compatibility with earlier versions of the compiler and many other UNIX systems; the `-G` option is the preferred specification. The `-Gn` and `-g` options disable all optimizations and imply `-O0`.

The debugging options take precedence over any conflicting options that appear on the command line. If more than one debugging option appears, the last one specified overrides the others.

2.17.2 `-h [no]bounds` (`cc` Command)

Default option: `-h nobounds`

The `-h bounds` option provides checking of pointer and array references to ensure that they are within acceptable boundaries. `-h nobounds` disables these checks.

The pointer check verifies that the pointer is greater than 0 and less than the machine memory limit. The array check verifies that the subscript is greater than or equal to 0 and is less than the array size, if declared.

2.17.3 `-h indef`, `-h zero` (`CC`, `cc` Commands)

The `-h indef` option causes stack-allocated memory to be initialized to undefined values. These values cause run-time errors to occur when an uninitialized stack variable is used, such as in a floating-point operation or in an array subscript. The `-h zero` option causes stack-allocated memory to be initialized to all zeros. These options are especially useful for debugging tasked codes.

2.18 Messages

The following sections describe compiler options that affect messages.

2.18.1 -h msglevel_*n* (CC, cc Commands)

Default option: -h msglevel_3

The -h msglevel_*n* option specifies the lowest level of severity of messages to be issued. Messages at the specified level and above are issued. Argument *n* can be 0 (comment), 1 (note), 2 (caution), 3 (warning), or 4 (error).

2.18.2 -h [no]message=*n*[: *n*...] (CC, cc Commands)

Default option: Determined by -h msglevel_*n*

The -h [no]message=*n*[:*n*...] option enables or disables specified compiler messages. *n* is the number of a message to be enabled or disabled. You can specify more than one message number; multiple numbers must be separated by a colon with no intervening spaces. For example, to disable messages CC-174 and CC-9, specify the following:

```
-h nomessage=174:9
```

The -h [no]message=*n* option overrides -h msglevel_*n* for the specified messages. If *n* is not a valid message number, it is ignored. Any compiler message except ERROR, INTERNAL, and LIMIT messages can be disabled; attempts to disable these messages by using the -h nomessage=*n* option are ignored.

2.18.3 -h report=*args* (CC, cc Commands)

The -h report=*args* option generates report messages specified in *args* and lets you direct the specified messages to a file. *args* can be any combination of the following:

args	Description
`i`	Generates inlining optimization messages
`m`	(Deferred implementation) Generates multi-streaming optimization messages
`s`	Generates scalar optimization messages
`t`	Generates tasking optimization messages
`v`	Generates vector optimization messages
`f`	Writes specified messages to file *file*`.V` where *file* is the source file specified on the command line. If the `f` option is not specified, messages are written to `stderr`.

No spaces are allowed around the equal sign (`=`) or any of the *args* codes. For example, the following example prints inlining and scalar optimization messages to file, `myfile.c`:

```
cc -h report=is myfile.c
```

2.18.4 `-h [no]abort` (`CC`, `cc` Commands)

Default option: `-h noabort`

The `-h [no]abort` option controls whether a compilation aborts if an error is detected.

2.18.5 `-h errorlimit[=n]` (`CC`, `cc` Commands)

Default option: `-h errorlimit=100`

The `-h errorlimit[=n]` option specifies the maximum number of error messages the compiler prints before it exits. *n* is a positive integer. Specifying `-h errorlimit=0` disables exiting on the basis of the number of errors. Specifying `-h errorlimit` with no qualifier is the same as setting *n* to `1`.

2.19 Compilation Phases

The following sections describe compiler options that affect compilation phases.

2.19.1 -E (CC, cc, c89, cpp Commands)

If the -E option is specified on the CC, cc, or c89 command lines, it executes only the preprocessor phase of the compiler. The -E and -P options are equivalent, except that -E directs output to stdout and inserts appropriate #line preprocessing directives. The -E option takes precedence over the -h feonly, -S, and -c options.

If the -E option is specified on the cpp command line, it inserts the appropriate #line directives in the preprocessed output. When both the -P and -E options are specified, the last one specified takes precedence.

2.19.2 -P (CC, cc, cpp Commands)

When the -P option is specified on the CC or cc command line, it executes only the preprocessor phase of the compiler for each source file specified. The preprocessed output for each source file is written to a file with a name that corresponds to the name of the source file and has .i suffix substituted for the suffix of the source file. The -P option is similar to the -E option, except that #line directives are suppressed, and the preprocessed source does not go to stdout. This option takes precedence over -h feonly, -S, and -c.

When the -P option is specified on the cpp command line, it is ignored. When both the -P and -E options are specified, the last one specified takes precedence.

2.19.3 -h feonly (CC, cc Commands)

The -h feonly option limits the C and C++ compilers to syntax checking. The optimizer and code generator are not executed. This option takes precedence over -S and -c.

2.19.4 -S (CC, cc Commands)

The -S option compiles the named C or C++ source files and leaves their assembly language output in the corresponding files suffixed with a .s. If this option is used with -G or -g, debugging information is not generated. This option takes precedence over -c.

2.19.5 -c (CC, cc, c89 Commands)

The -c option creates a relocatable object file for each named source file, but does not link the object files. The relocatable object file name corresponds to the

name of the source file. The .o suffix is substituted for the suffix of the source file.

2.19.6 -#, -##, and -### (CC, cc, cpp Commands)

The -# option produces output indicating each phase of the compilation as it is executed. Each succeeding output line overwrites the previous line.

The -## option produces output indicating each phase of the compilation, as well as all options and arguments being passed to each phase, as they are executed.

The -### option is the same as -##, except the compilation phases are not executed.

2.19.7 -W*phase*["*opt*..."] (CC, cc Commands)

The -W *phase* option passes arguments directly to a phase of the compiling system. The -W option appears with argument *phase* to indicate system phases as follows:

phase	System phase	Command
p	Preprocessor	
0	Compiler	
a	Assembler	as on Cray PVP systems, cam on Cray MPP systems
l	Loader	System-specific; ld or cld.

Arguments to be passed to system phases can be entered in either of two styles. If spaces appear within a string to be passed, the string is enclosed in double quotes. When double quotes are not used, spaces cannot appear in the string. Commas can appear wherever spaces normally appear; an option and its argument can be either separated by a comma or not separated. If a comma is

part of an argument, it must be preceded by the \ character. For example, any of the following command lines would send `-e name` and `-s` to the loader:

```
cc -Wl,"-e name -s" file.c

cc -Wl,-e,name,-s file.c

cc -Wl,"-ename",-s file.c
```

Because the preprocessor is built into the compiler, `-Wp` and `-W0` are equivalent.

2.19.8 -Y*phase*,*dirname* (CC, cc, c89, cpp Commands)

The `-Y`*phase*,*dirname* option specifies a new directory (*dirname*) from which the designated *phase* should be executed. *phase* can be one or more of the following values:

phase	System phase	Command
p	Preprocessor	
0	Compiler	
a	Assembler	`as` on Cray PVP systems, `cam` on Cray MPP systems
l	Loader	System-specific; `ld` or `cld`

Because there is no separate preprocessor, `-Yp` and `-Y0` are equivalent. If you are using the `-Y` option on the `cpp` command line, `p` is the only argument for *phase* that is allowed.

2.20 Preprocessing

The following sections describe compiler options that affect preprocessing.

2.20.1 -I *incldir* (CC, cc, c89, cpp Commands)

The `-I` *incldir* option specifies a directory for files named in `#include` directives when the `#include` file names do not have a specified path. Each directory specified must be specified by a separate `-I` option.

The order in which directories are searched for files named on `#include` directives is determined by enclosing the file name in either quotation marks (`" "`) or angle brackets (`<>`).

Directories for `#include "`*file*`"` are searched in the following order:

1. Directory of the input file.
2. Directories named in `-I` options, in command-line order.
3. Site- and compiler release-specific include files directories.
4. Directory `/usr/include`.

Directories for `#include <`*file*`>` are searched in the following order:

1. Directories named in `-I` options, in command-line order.
2. Site-specific and compiler release-specific include files directories.
3. Directory `/usr/include`.

If the `-I` option specifies a directory name that does not begin with a backslash (`/`), the directory is interpreted as relative to the current working directory and not relative to the directory of the input file (if different from the current working directory). For example:

```
cc -I. -I yourdir mydir/b.c
```

The preceding command line produces the following search order:

1. `mydir` (`#include "`*file*`"` only).
2. Current working directory, specified by `-I.`
3. `yourdir` (relative to the current working directory), specified by `-I yourdir`.
4. Site-specific and compiler release-specific include files directories.
5. Directory `/usr/include`.

2.20.2 `-D` *macro*`[=`*def*`]` (`CC`, `cc`, `c89`, `cpp` Commands)

The `-D` *macro*`[=`*def*`]` option defines a macro named *macro* as if it were defined by a `#define` directive. If no `=`*def* argument is specified, *macro* is defined as `1`.

Predefined macros also exist; these are described in Chapter 6, page 99. Any predefined macro except those required by the standard (see Section 6.1, page 99) can be redefined by the -D option. The -U option overrides the -D option when the same macro name is specified regardless of the order of options on the command line.

2.20.3 -U *macro* (CC, cc, c89, cpp Commands)

The -U option removes any initial definition of *macro*. Any predefined macro except those required by the standard (see Section 6.1, page 99) can be undefined by the -U option. The -U option overrides the -D option when the same macro name is specified, regardless of the order of options on the command line.

Predefined macros are described in Chapter 6, page 99. Macros defined in the system headers are not predefined macros and are not affected by the -U option.

2.20.4 -M (CC, cc, cpp Commands)

The -M option provides information about recompilation dependencies that the source file invokes on #include files and other source files. This information is printed in the form expected by make(1). Such dependencies are introduced by the #include directive. The output is directed to stdout.

2.20.5 -N (cpp Command)

The -N option specified on the cpp command line enables the old style (referred to as K & R) preprocessing. If you have problems with preprocessing (especially non-C source code), use this option.

2.20.6 -C (CC, cc, cpp Commands)

The -C option retains all comments in the preprocessed source code, except those on preprocessor directive lines. By default, the preprocessor phase strips comments from the source code. This option is useful with cpp or in combination with the -P or -E option on the CC and cc commands.

2.20.7 -h [no]pragma=*name*[: *name*...] (CC, cc Commands)

Default option: -h pragma

The [no]pragma=*name*[:*name*...] option enables or disables the processing of specified directives in the source code. *name* can be the name of a directive

or a word shown in Table 2 to specify a group of directives. More than one name can be specified. Multiple names must be separated by a colon and have no intervening spaces.

Table 2. `-h pragma` directive processing

name	Group	Directives affected
`all`	All	All directives
`allinline`	Inlining	`inline`, `noinline`
`allscalar`	Scalar optimization	`align`, `cache_align`, `cache_bypass`, `concurrent`, `nointerchange`, `noreduction`, `split`, `suppress`, `unroll`
`alltask`	Tasking	`case`, `endcase`, `guard`, `endguard`, `taskloop`, `endloop`, `parallel`, `endparallel`, `prefertask`, `taskcommon`, `common`, `taskprivate`, `taskshared`
`allvector`	Vectorization	`ivdep`, `novector`, `novsearch`, `prefervector`, `shortloop`

When using this option to enable or disable individual directives, note that some directives must occur in pairs (for example, `parallel` and `endparallel`). For these directives, you must disable both directives if you want to disable either; otherwise, the disabling of one of the directives may cause errors when the other directive is (or is not) present in the compilation unit.

2.21 Loader

The following sections describe compiler options that affect loader tasks.

2.21.1 `-l` *libfile* (`CC`, `cc`, `c89` Commands)

The `-l` *libfile* option identifies library files to be loaded. If *libfile* begins with a period (`.`) or slash (`/`), it is assumed to be a path name and is used without modification. An initial `.` (or `..`) is interpreted as the current working

directory (or its parent directory). It is not relative to the input file's directory if that differs from the current working directory.

There is no search order dependency for libraries. Default libraries are shown in the following list.

`libC.a` (C++ only)

`libu.a`

`libm.a`

`libc.a`

`libsma.a` (UNICOS/mk systems only)

`libf.a`

`libfi.a`

`libsci.a`

If you specify personal libraries by using the `-l` command-line option, as in the following example, those libraries are added to the top of the preceding list. (The `-l` option is passed to the loader.)

```
cc -l mylib target.c
```

When the previous command line is issued, the loader looks for a library named `libmylib.a` (following the naming convention) and adds it to the top of the list of default libraries.

2.21.2 `-L` *libdir* (`CC`, `cc`, `c89` Commands)

The `-L` *libdir* option changes the `-l` option algorithm to search directory *libdir* before searching the default directories. If *libdir* does not begin with a slash (`/`), it is interpreted as relative to the current working directory.

The loader searches for library files in the default directories in the following order:

1. Site-specific and compiler release-specific library directories
2. `/lib`
3. `/usr/lib`

Note: Multiple `-L` options are treated cumulatively as if all *libdir* arguments appeared on one `-L` option preceding all `-l` options. Therefore, do not attempt to load functions of the same name from different libraries through the use of alternating `-L` and `-l` options.

If the `-F` loader option is used to include the system default directories, the loader searches directory *libdir* for those libraries before searching the default directories.

2.21.3 `-o` *outfile* (`CC`, `cc`, `c89` Commands)

The `-o` *outfile* option produces an absolute binary file named *outfile*. A file named `a.out` is produced by default. When this option is used in conjunction with the `-c` option and a single C or C++ source file, a relocatable object file named *outfile* is produced.

2.21.4 `-d` *string* (`CC`, `cc` Commands)

The `-d` *string* option specifies a character string comprised of directive names separated by semicolons that is sent to the loader to be inserted into the loader directives file and processed as though the `-D` *dirstring* option had been specified on the loader command. This allows you to manipulate the loader while using the compiler command.

2.21.5 `-s` (`CC`, `cc`, `c89` Commands)

The `-s` option produces executable files from which symbolic and other information not required for proper execution has been removed. If both the `-s` and `-g` (or `-G`) options are present, `-s` is ignored.

2.22 General

The following sections describe compiler options that affect general tasks.

2.22.1 `-V` (`CC`, `cc`, `cpp` Commands)

The `-V` option displays compiler version information. If the command line specifies no source file, no compilation occurs. Version information consists of the product name, the version number, and the current date and time, as shown in the following example:

```
% CC -V
Cray C++ Version 3.1.0  07/25/98  18:31:43
```

If a file is specified, information about the compilation is displayed in addition to the version information. The additional information includes the compiler generation date, the compilation execution time, the maximum memory used by the compiler (in decimal words), and the resulting number of words used for code and data.

2.22.2 `-X` *npes* (`CC`, `cc` Commands)

(UNICOS/mk systems) The `-X` *npes* option specifies how many processing elements (PEs) are to be used on Cray T3E systems. The *npes* argument specifies the number of PEs and has no default value, it must be explicitly set. For the *npes* argument, specify either an integer from 1 through 2048 or `m`. A value of `m` directs the compiler to generate a malleable `a.out` file. Specifying `-X m` allows you to change the number of PEs used each time the executable `a.out` file is run. If you specify `-X m`, use the `mpprun`(1) command and its `-n` option to specify the number of PEs you want to use. For more information, see `mpprun`(1). If you do not use `mpprun`(1) on the `a.out` file that is generated when `-X m` is specified, the operating system executes the file on a single processor just as if you had invoked `mpprun`(1) with one processor.

The option is passed from the command to both the compiler and the loader. If the compiler recognizes the option, it becomes a compile-time value and cannot be changed at load time. If the loader recognizes the option, it is a load-time value and cannot be changed at `mppexec`(1) time. For example:

```
cc -X8 -c file.c
cc -X8 file.c
```

In these cases, the value 8 is set at compile time only. In the first line, the loader is not called (specified by the `-c` option) and the option is passed only to the compiler. In the second line, the option is passed to both the compiler and the loader, but since it is first recognized by the compiler it is a compile-time constant, not a load-time constant.

In the following case, the value 8 is used at load time only. The compiler is not called because no source file is specified and the option is passed only to the loader.

```
cc -X8 file.o
```

The -W option can also be used to specify which phase of compilation gets the -X option. For example:

```
cc -W0,-X8 file.c
cc -Wl,-X8 file.c
```

In the first line, the -X option is passed only to the compiler, and the number of PEs is set to 8 at compile time. In the second line, the -X option is passed only to the loader and the number of PEs is set to 8 at load time.

If the number of PEs is specified at both compile and load time, the compile-time constant overrides the load-time constant. If the two values disagree, the loader issues a message.

2.22.3 -h ident=*name* (CC, cc Commands)

Default option: File name specified on the command line

The -h ident=*name* option changes the ident name to *name*. This *name* is used as the module name in the object file (.o suffix) and assembler file (.s suffix). Regardless of whether the ident name is specified or the default name is used, the following transformations are performed on the ident name:

- All . characters in the ident name are changed to $.
- If the ident name starts with a number, a $ is added to the beginning of the ident name.

2.23 Command-line Examples

The following examples illustrate a variety of CC and cc command lines.

- The following example compiles myprog.C on UNICOS/mk systems, fixing the number of processing elements (PEs) to 8 and instantiating all template entities that are declared or referenced in the compilation unit.

  ```
  CC -X8 -h instantiate=all myprog.C
  ```

- The following example compiles myprog.C. The -h conform option specifies strict conformance to the draft ISO C++ standard. No automatic instantiation of templates is performed.

  ```
  CC -h conform -h noautoinstantiate myprog.C
  ```

- The following example compiles input files myprog.C and subprog.C. Option -c specifies that object files myprog.o and subprog.o are produced and that the loader is not called. Option -h inline1 instructs the compiler to inline function calls.

```
CC -c -h inline1 myprog.C subprog.C
```

- The following example specifies that the compiler search the current working directory (represented by a period (.)) for #include files before searching the default #include file locations.

```
CC -I. disc.C vend.C
```

- The following example specifies that source file newprog.c be preprocessed only. Compilation and linking are suppressed. In addition, the macro DEBUG is defined.

```
cc -P -D DEBUG newprog.c
```

- The following example compiles mydata1.C, writes object file mydata1.o, and produces a scalar optimization report to stdout.

```
CC -c -h report=s mydata1.C
```

- The following example compiles mydata3.c and produces the executable file a.out. A 132-column pseudo assembly listing file is also produced in file mydata3.L.

```
cc -h listing mydata3.c
```

- The following example compiles myfile.c and passes an option to the loader (-Dalign=modules) that causes blocks of code to be aligned.

```
cc -Wl,"-Dalign=modules" myfile.c
```

- The following example compiles myfile.c and instructs the compiler to generate additional run-time code needed to support use of the ATExpert system. A tasking report is directed to stdout.

```
cc -h atexpert,report=t myfile.c
```

- The following example compiles myfile.C and instructs the compiler to attempt to inline calls aggressively to functions defined within myfile.C. An inlining report is directed to myfile.V.

```
CC -h inline3,report=if myfile.C
```

2.24 Environment Variables

The environment variables listed below are used during compilation.

Variable	Description
CRI_CC_OPTIONS, CRI_cc_OPTIONS, CRI_c89_OPTIONS, CRI_cpp_OPTIONS	Specifies command-line options that are applied to all compilations. Options specified by this environment variable are added following the options specified directly on the command line. This is especially useful for adding options to compilations done with build tools.
LANG	Identifies your requirements for native language, local customs, and coded character set with regard to compiler messages.
MSG_FORMAT	Controls the format in which you receive compiler messages.
NLSPATH	Specifies the message system catalogs that should be used.
NPROC	Specifies the number of processes used for simultaneous compilations. The default is 1. When more than one source file is specified on the command line, compilations may be multiprocessed by setting the environment variable NPROC to a value greater than 1. You can set NPROC to any value; however, large values can overload the system.
TARGET	Specifies type and characteristics of the hardware on which you are running. You can also set the TARGET environment variable to the characteristics of another Cray Research system to cross-compile source code for that system. See the target and sh man pages for more information.

Cray C and C++ Extensions [3]

This chapter describes the Cray Standard C and Cray C++ extensions to their respective standards. A program that uses one or more extensions does not strictly conform to the standard. These extensions are not available in strict conformance mode.

Extensions to both the C and C++ standards include the following:

- Restricted pointers (Section 3.1, page 43)
- `long long` types (Section 3.2, page 47)
- `//` comments (Section 3.3, page 48)

The following extensions apply only to the C standard:

- Complex data types (Section 3.4, page 48)
- Variable length arrays (Section 3.5, page 50)
- `fortran` keyword (Section 3.6, page 57)
- Hexadecimal floating-point constants (Section 3.7, page 58)

3.1 Restricted Pointers

In extended mode, the identifier `restrict` is treated as a keyword that specifies a third type qualifier (in addition to `const` and `volatile`). A pointer with `restrict`-qualified type is termed a ***restricted pointer***.

Restricted pointers are a language extension offered by Cray Standard C and Cray C++ for asserting the absence of aliasing through pointers. One advantage of using restricted pointers on UNICOS systems is that they can increase the number of loops that the compiler can vectorize. On UNICOS/mk systems, they can improve instruction scheduling and memory hierarchy analysis.

When it is inconvenient to modify the source code so that the `restrict` qualifier appears explicitly in pointer declarations, use the `-h restrict=` command-line option (for more information, see Section 2.7.2, page 10).

A program that uses the `restrict` qualifier can easily be ported to a system that does not support the qualifier by defining the identifier `restrict` to be an empty preprocessor macro. You can do this with the `-D restrict=` option or by using the following code:

```
#ifndef _CRAYC #define restrict #endif
```

If the preceding directives are included in a program, a declaration of the following form expands into a declaration of a restricted pointer with Cray Research compilers and into a declaration of an ordinary pointer with other compilers:

```
int * restrict p;
```

On UNICOS systems, the compiler ignores some uses of `restrict` that it currently cannot vectorize. Some cases that are ignored are discussed in Section 3.1.6, page 46.

3.1.1 Function Parameters

If a function has pointer parameters that point into disjoint (array or non-array) objects in each call, the use of these parameters can be asserted to the compiler by declaring the pointer parameters to be `restrict`-qualified. This eliminates potential aliasing through those pointers as an obstacle to vectorization of the loops in the function, and also allows improved scheduling of scalar code.

The following example illustrates the use of `restrict`-qualified pointer parameters:

```
void f1 (int n, int * restrict p, int * restrict q) {
   int i;

   /* Compiler may analyze dependences  */
   /* as if the following two lines     */
   /* were present:                     */
   /*   p = malloc(n * sizeof(int));    */
   /*   q = malloc(n * sizeof(int));    */

  for (i = 0; i < n; i++) { /* Vectorized   */
        p[i] = q[i];
  }
}
```

This operation will also vectorize if the restricted pointers are incremented as in the following example:

```
void f2 (int n, int * restrict p, int * restrict q) {
   int i;
   for (i = 0; i < n; i++) { /* Vectorized */
        *p++ = *q++;
   }
}
```

3.1.2 File Scope

Restricted pointers are also useful when declared at file scope. In the next example, `restrict` is used to assert that the global pointer `p` points into a unique array object, which in this case is obtained from a `malloc` call.

```
#include <stdlib.h>

int * restrict p;

void alloc_p(int n) {
   p = malloc(n * sizeof p[0]);
}
```

3.1.3 Block Scope

Similarly, a restricted pointer declared in the outermost block of a function can be used to point to an array allocated for use within the function. Continuing the example shown in the previous subsection:

```
extern int * restrict p;

void modify_p(int n) {
   int * restrict q = malloc(n * sizeof q[0]);
   int i;
   for (i = 0; i < n; i++) {    /* Vectorized */
      q[i] = p[i];
   }
   for (i = 0; i < n-1; i+=2) { /* Vectorized */
      p[i]   = q[i+1] + q[i];
      p[i+1] = q[i+1] - q[i];
   }
   free(q);
}
```

3.1.4 Unrestricted Pointers

Like array names, restricted pointers are not immune from aliasing through unrestricted pointers. At the point of its declaration, a restricted pointer may be assumed to provide the only means of designating an array object that is disjoint from any other object, but a restricted pointer's value can be assigned to one or more unrestricted pointers. Such an assignment may occur in an assignment expression or in the assignment of an argument value to a parameter in a function call. Although permitted, the use of restricted and unrestricted pointers together should be avoided, since such use often inhibits vectorization.

3.1.5 Comparison with `#pragma ivdep`

Restricted pointers and the `#pragma ivdep` directive both provide a means of promoting vectorization. Because they convey different information to the compiler, they can give different results. One may be effective in causing a particular loop to vectorize when the other is not. For loops that will vectorize either way, restricted pointers may give better performance.

3.1.6 Implementation Limits

At default vectorization level, the current implementation only takes advantage of the qualifier for simple identifiers declared with file scope, as function parameters, or in the outermost block of a function. In other contexts, uses of the qualifier are checked only for syntactic correctness. The `-h vector3`

option enables a more detailed analysis that can take advantage of restricted pointers that are members of structures or elements of arrays.

Inlining a function with restricted pointer parameters can cause a loss of vectorization.

3.2 long long and unsigned long long Data Types

Cray Standard C and Cray C++ provide long long and unsigned long long data types, which are supported on all Cray Research systems. long long and unsigned long long are new, 64-bit integral types that are identical in format to long and unsigned long, respectively, and are provided for compatibility with C and C++ compilers supplied by other vendors. These types are not available if you are compiling in strict conformance mode.

Note: long long and unsigned long long were available prior to the 6.1 release of the Cray Standard C compiler, but they were a synonym for long and unsigned long, respectively. In the 6.1 release, long long and unsigned long long are separate integral types.

A long long constant is represented like a long constant, except the suffix is ll. An unsigned long long constant is represented like an unsigned long constant, except the suffix is llu or ull.

Adding long long and unsigned long long types to the C language affects the usual arithmetic conversions as specified in the standard. See Section 3.8, page 59, for information on how these types affect the performance of usual arithmetic conversions on Cray machines.

The following changes have also been made to the Standard C library and header files to support the long long and unsigned long long types. These changes are supported on the UNICOS 10.0.0.2 and UNICOS/mk 2.0.4 and later operating systems.

- The LLONG_MAX, LLONG_MIN, and ULLONG_MAX macros have been added to limits.h.
- The llabs, lldiv, strtoll, atoll, strtoull functions have been added to the C library and are declared in stdlib.h.
- The wcstoll and wcstoull functions have been added to the C library and are declared in wchar.h.

- The printf family of functions allows an ll specifying that a following d, i, o, u, x, or X conversion specifier applies to a long long or unsigned long long argument. They also allow an ll specifying that a following n conversion specifier applies to a pointer to a long long argument. Similarly, the scanf family of functions allow an ll specifying that a following d, i, or n conversion specifier applies to a pointer to long long.
- The printf family of functions allows an ll specifying that a following d, i, o, u, x, or X conversion specifier applies to a long long int or unsigned long long int argument. They also allow an ll specifying that a following n conversion specifier applies to a pointer to a long long int argument. Similarly, the scanf family of functions allows an ll specifying that a following d, i, or n conversion specifier applies to a pointer to long long int.

3.3 // Comments

In extended mode, Cray Standard C allows C++ style // comment lines. The // characters start a comment, which terminates with the next newline character.

If there is a form-feed or a vertical-tab character in this type of comment, only white space characters can appear between it and the newline character that terminates the comment. The //, /*, and */ comment characters have no special meaning within a // comment, and they are treated like other characters. Similarly, the // and /* comment characters have no special meaning within a /* comment.

3.4 Complex Data Types

Note: Complex data types are available in Cray Standard C only. Complex data types are supported in Cray C++ through the complex class library.

Cray Standard C provides float complex, double complex, and long double complex data types for use on all Cray systems. These data types are available only if the nonstandard header <complex.h> is included in your source. These data types are available even if you are compiling in strict conformance mode, as long as the <complex.h> header is included. Complex arithmetic can be performed in much the same way as with real data types (either integral or floating type). Complex variables can be declared and initialized. Most arithmetic operators can be used with one or two complex operands to yield a complex result. Many standard math functions have

corresponding functions that take complex arguments and return complex values.

The complex data types are represented in memory with two contiguous parts, the *real* part and the *imaginary* part. The characteristics of the imaginary part agree with those of the corresponding real types. For example, the imaginary part of a `float complex` type has the same characteristics as a `float`.

An imaginary constant has the following form:

R`i`

R is either a floating-constant or an integer-constant; no space or other character can appear between *R* and `i`. If you are compiling in strict conformance mode (`-h conform`), imaginary constants are not available.

3.4.1 Complex Usage

A complex variable is initialized by using an expression that may contain an imaginary constant. For example:

```
#include <complex.h>
double complex z1 = 1.2 + 3.4i;
double complex z2 = 5i;
```

When the `++` operator is used to increment a complex variable, only the real part is incremented.

Printing a complex value requires the use of two `%f` specifications and any formatting needed for legibility must be specified as shown in the following example:

```
double complex z = 0.5 + 1.5i;
printf("<.2f,%.2f>\n ", creal(z), cimag(z));
```

The output from the preceding example is as follows:

```
<0.50,1.50>
```

3.4.2 Conversion to and from Complex

A binary operator with one complex operand and one real operand causes the real operand to be promoted to a complex type before the operation is performed. When a real value is promoted to a complex value, the new complex value's real part gets the same value as if the promotion were to the

corresponding floating type, and its imaginary part is zero. When a complex value is demoted to a real type, the value of its imaginary part is discarded, and its real part is demoted according to the demotion rules for the corresponding floating type.

A complex type is not valid as an operand for any operator that requires an integral data type. A complex type is not valid as an operand for any of the relational operators, but it is valid for the equality and inequality operators. It is valid as an operand for any other operator that allows floating data types.

Math functions that take complex arguments are declared in the `complex.h` header and described in the *UNICOS System Libraries Reference Manual*.

3.4.3 Arithmetic Conversion for Complex

Adding complex types to the C language affects the usual arithmetic conversions as specified in the standard. See Section 3.8, page 59 for information on how complex types affect how the usual arithmetic conversions are performed on Cray machines.

3.5 Variable Length Arrays

Note: Variable length arrays are not supported in Cray C++.

In extended mode, you can declare variable length arrays. A *variable length array* (VLA) is an array that has a size (at least one dimension) that is determined at run time. The ability to use variable length arrays enhances the compiler's range of use for numerical programming.

3.5.1 Declarator Restrictions

An array type with a size specifier (dimension) that must be evaluated at program execution time is said to be a *variable length array type*.

Use of this variable type is restricted to only block and function prototype scopes. An object type that contains one or more derived declarator types of variable length array type is said to be *variably modified*. For example, a "pointer to a VLA" is a pointer (not a VLA) but it is still variably modified. Members of structure and union types are not allowed to have variably modified type.

Function parameters can be declared with variable length array type. For example, the following `matrix_mult` function declaration can be used to

define a function that performs a matrix multiply of an n by m and an m by n matrix to yield an n by n result:

```
void matrix_mult(int n, int m, double a[n][n],
                 double b[n][m], double c[m][n]);
```

In the previous example, the array sizes are computed each time the function is called. In addition, the conventional workaround of maintaining a table of pointers to each of the rows in the matrices is avoided because variable array addressing is done automatically.

C variable length arrays are similar to Fortran automatic and adjustable arrays. Variable length arrays can also be used as local (auto) variables. Previously, a call to the malloc(3) library function was required to allocate such dynamic arrays, and a call to the free(3) library function was required to deallocate them.

3.5.2 Variable Length Array (VLA) Declarators

The size of a variable length array (VLA) dimension must be of integer type and is not a constant expression. The only storage class specifiers that can be explicitly given are auto (for block scope arrays) and register (for parameter array declarators). Using an expression that produces side effects (for example, function calls) to specify the size of an array is permitted, however, the order of evaluation of two or more such expressions within a single declaration is undefined.

A variable length array declaration cannot have an initializer.

For two array types to be compatible, both must have compatible element types, and if both size specifiers are present and are integer constant expressions, then both sizes must have the same value. A VLA is always compatible with another array type if they both have the same element type. If the two array types are used in a context that requires them to be compatible, it is undefined behavior if the dimension sizes are unequal at run time.

The following example illustrates arrays that are incompatible and arrays that are compatible but have undefined results unless certain criteria are met:

```
extern int n;
extern int m;
void func(void)
{
  int a[n][6][m];
  int (*p)[4][n];        /* pointer to VLA */
  int c[n][n][6][m];
  int (*r)[n][n][n+1];   /* pointer to VLA */

  p = a;/* error - not compatible because 4 != 6 */

  r = c; /* compatible - but undefined behavior
            unless n==6 and m==n+1 */
}
```

The following example illustrates arrays that are compatible, but have undefined behavior at execution time:

```
int dim4 = 4;
int dim6 = 6;
main()
{
  int (*q)[dim4][8][dim6]; /* pointer to VLA */
  int (*r)[dim6][8][1];    /* pointer to VLA */

  r = q; /* compatible, but undefined behavior at
            execution time */
}
```

3.5.3 Function Declarators and Variable Length Arrays

For each parameter declared with variable length array type, the type used for compatibility comparisons is the one that results from conversion to a pointer type, as for fixed length arrays.

All identifiers used in VLA size expressions must be declared prior to use (as is the case with the usage of all variables). Thus, the order in which function parameters are declared is important when VLAs are used. For example:

```
void f1(int n1, double a1[n1]) {} /* Correct */

void f2(int a2[n2], int n2) {}    /* Error: n2 not declared before VLA */
```

```
int n3;
void f3(int a3[n3], int n3) {}     /* Correct, but file scope n3 is used in VLA
                                      size expression */
```

The manner in which a function declaration with variably modified parameters is handled depends on whether the function definition is found, as follows:

- If the function definition is found, all size expressions of the variably modified parameters are evaluated on entry to the function.
- If no function definition is found, the array modifier types affected do not need to be completed, and the associated dimension size specifier expressions are not evaluated. If the associated size specifier expressions are present, however, any identifier used to specify the size of these variable length array types needs to be visible. For example:

  ```
  void f(double a[x][y]);  /* Error: x and y are not declared */
  ```

  For prototype declarators that are not definitions, an alternative method of specifying a variable length array modifier is to use the `[*]` syntax notation to indicate that the array modifier type does not need to be completed. For example:

  ```
  void f(int, int, int[*][*]);
  ```

  The following example shows compatible function declarators used as prototypes:

  ```
  void matrix_mult(int n, int m, double a[n][m],
                   double b[m][n],double c[n][n]);

  void matrix_mult(int n, int m, double a[*][*],
                   double b[*][*], double c[*][*]);

  void matrix_mult(int n, int m, double a[][*],
                   double b[][*], double c[][*]);
  ```

3.5.4 Variable Length Array Type Definitions

Type definition declarations (that is, `typedef` names) that specify a variable length array (VLA) type must have block scope. The size of the VLA is determined at the time that the type definition is declared and not at the time it is invoked for use in an actual declarator.

The following example shows the use of a VLA type definition:

```
void func(int n)
{
     typedef int A[n]; /* Correct; declared in block scope */
     A a;
     A *p = &a;
}
```

The following example shows an invalid VLA type definition:

```
int n;
typedef int A[n]; /* not valid; declared in file scope */
```

The following example declares VLAs at different scopes:

```
void func(int n)
{
     typedef int A[n]; /* A is n ints; current value of n */
     n += 1;
     {
        A a; /* a is n ints; n without += 1 */
        int b[n];/* a and b are different sizes  */
        for (i = 1; i < n; i++)
           a[i-1] = b[i];
     }
}
```

3.5.5 `sizeof` Operator and Variable Length Arrays

When the `sizeof` operator is applied to an operand that has variable length array type, the result is not a constant expression (unlike other `sizeof` expressions) and is computed at program execution time. For example:

```
int func(int n) {
    char b[n];
    return sizeof(b);
}

main() {
    printf("%d\n", func(10));
}
```

The output from the preceding example is `10`.

3.5.6 `goto` Statements

Use of a `goto` statement to jump into a block scope level where a variably qualified object has been declared is not allowed. For example:

```
void func(int n)
{
     int j = 4;
     goto lab3; /* error - going INTO scope of VLA */
     {
        double a[n];
        a[j] = 4.4;
lab3:
        a[j] = 3.3;
        goto lab4; /* OK - going WITHIN scope of VLA */
        a[j] = 5.5;
lab4:
        a[j] = 6.6;
     }
     goto lab4; /* error - going INTO scope of VLA */
     return;
  }
```

3.5.7 `switch` Statement

When using the `switch` statement, the controlling expression must not cause control to jump into a block where a variably qualified object has been declared.

```
int i, j = 30;
main() {
   int n = 10;
   int m = 20;
   switch (n) {
     int a[n]; /*error; switch bypasses declaration of a[n] */
     case 10:
        a[0] = 1;
        break
     case 20:
        a[0] = 2;
        break;
   }
   switch (i) {
     case 0:
        {
           int b[n]; /*OK; declaration of b[n] not bypassed */
           b[2] = 4;
        }
        break;
     case 1:
        break;
   }
}
```

3.5.8 `setjmp` and `longjmp` Functions

If the `longjmp` function returns control back to the point of the `setjmp` invocation, the memory associated with a variable length array object could be squandered.

In the following example, the function `h` causes storage to be lost for the variable length array object `a` which is declared in function `g`, because its existence is unknown to `h`. Additional storage would be lost for variable length array object `b`, which is declared in function `h`.

```
#include <setjmp.h>
void g(int n);
void h(int n);
int n = 6;
jmp_buf buf;

void f(void) {
  int x[n];
  setjmp(buf);
  g(n);
}

void g(int n) {
  int a[n];
  h(n);
}

void h(int n) {
  int b[n];
  longjmp(buf,2);
}
```

3.6 `fortran` Keyword

Note: The `fortran` keyword is not allowed in Cray C++.

In extended mode, the identifier `fortran` is treated as a keyword. It specifies a storage class that can be used to declare a Fortran-coded external function. The use of the `fortran` keyword when declaring a function causes the compiler to verify that the arguments used in each call to the function are pass-by-address; any arguments that are not addresses are converted to addresses.

As in any function declaration, an optional ***type-specifier*** declares the type returned, if any. Type `int` is the default; type `void` can be used if no value is returned (by a Fortran subroutine). The `fortran` storage class causes conversion of lowercase function names to uppercase, and, if the function name ends with an underscore character, the trailing underscore character is stripped from the function name. (Stripping the trailing underscore character is in keeping with UNIX practice.)

Functions specified with a `fortran` storage class must not be declared elsewhere in the file with a `static` storage class.

An example using the `fortran` keyword is shown in Section 8.1.2.7, page 116.

3.7 Hexadecimal Floating-point Constants

Note: Hexadecimal floating-point constants are not available in Cray C++.

Floating constants can be represented in hexadecimal format. This feature is not portable, because identical hexadecimal floating constants can have different meanings on different systems.

A hexadecimal floating constant can be used whenever traditional floating-point constants are allowed.

The hexadecimal constant has the usual syntax: `0x` (or `0X`) followed by hexadecimal characters. The optional floating suffix has the same form as for normal floating constants: `f` or `F` (for float), `l` or `L` (for long), optionally followed by an `i` (imaginary).

The constant must represent the same number of bits as its type, which is determined by the suffix (or the default of double). The constant's bit length is four times the number of hexadecimal digits, including leading zeros.

The following example illustrates hexadecimal constant representation:

`0x7f7fffff.f`	32-bit float
`0x0123456789012345.`	64-bit double

The value of a hexadecimal floating constant is interpreted as a value in the specified floating type. This uses an unsigned integral type of the same size as the floating type, regardless of whether an object can be explicitly declared with such a type. No conversion or range checking is performed. The resulting floating value is defined in the same way as the result of accessing a member of floating type in a union after a value has been stored in a different member of integral type.

The following example illustrates hexadecimal floating-point constant representation on UNICOS systems that use Cray floating-point format:

```
float f=0x3ffe800000000000.f;
double g=0xffffffffffffffff.;
main()
{
     printf("f = 0x%16x.f == %g\n", f, f);
     printf("g = 0x%16x. == %g\n", g, g);
}
```

The output from the preceding code is as follows:

```
f = 0x3ffe800000000000.f == 0.125
g = 0xffffffffffffffff. == *.00000
```

3.8 Arithmetic Conversions

Adding `complex` and `long long` types to the C and C++ languages affects the usual arithmetic conversions as specified in the standard. As a result, conversions are performed as follows on Cray Research machines:

1. If one operand is `long double complex`, the other operand is converted to `long double complex`, and the result is `long double complex`.
2. Else, if one operand is `float complex` or `double complex` and the other is `long double`, both are converted to `long double complex` and the result is `long double complex`.
3. Else, if either operand is `double complex`, the other operand is converted to `double complex`, and the result is `double complex`.
4. Else, if one operand is `float complex` and the other is `double`, both operands are converted to `double complex`, and the result is `double complex`.
5. Else, if either operand is `float complex`, the other operand is converted to `float complex`, and the result is `float complex`.
6. Else, if either operand is `long double`, the other is converted to `long double`, and the result is `long double`.
7. Else, if either operand is `double`, both are converted to `double`, and the result is `double`.
8. Else, if either operand is `float`, both are converted to `float`, and the result is `float`.
9. Else, if the integral promotions are performed on both operands. Then:
    a. If either operand has type `unsigned long long int`, the other operand is converted to `unsigned long long int`, and the result is `unsigned long long int`.
    b. Else, if one operand has type `long long int` and the other has type `unsigned long int`, both operands are converted to

`unsigned long long int` and the result is `unsigned long long int`.

c. Else, if one operand has type `long long int` and the other has type `unsigned int`, both operands are converted to `unsigned long long int` and the result is `unsigned long long int`.

d. Else, if either operand has type `long long int`, the other operand is converted to `long long int`, and the result is `long long int`.

e. Else, if either operand is `unsigned long`, both are converted to `unsigned long`, and the result is `unsigned long`.

f. Else, if one operand has type `long int`, and the other has type `unsigned int`, both operands are converted to `unsigned long` and the result is `unsigned long`.

g. Else, if either operand is `long`, the other is converted to `long`, and the result is `long`.

h. Else, if either operand is `unsigned int`, the other is converted to `unsigned int` and the result is `unsigned int`.

i. Else, both operands must be `int`, and the result is `int`.

#pragma Directives [4]

#pragma directives are used within the source program to request certain kinds of special processing. #pragma directives are part of the C and C++ languages, but the meaning of any #pragma directive is defined by the implementation. #pragma directives are expressed in the following form:

```
#pragma [ _CRI] identifier [arguments]
```

The _CRI specification is optional and ensures that the compiler will issue a message concerning any directives that it does not recognize. Diagnostics are not generated for directives that do not contain the _CRI specification.

These directives are classified according to the following types:

- Loop
- General
- Instantiation (C++ only)
- Vectorization
- Scalar
- Tasking
- Inlining

Macro expansion occurs on the directive line after the directive name. (That is, macro expansion is applied only to *arguments*.) For example, if NUM_CHUNKS is a macro defined as the value 8, the original code is as follows:

```
#pragma _CRI taskloop numchunks(NUM_CHUNKS)
```

The expanded code is equivalent to the following:

```
#pragma _CRI taskloop numchunks(8)
```

At the beginning of each section that describes a directive, information is included about the compilers and systems that allow the use of the directive, and the scope of the directive. Unless otherwise noted, the following default information applies to each directive:

Compiler:	C and C++
Operating System:	UNICOS and UNICOS/mk
Scope:	Local and global

4.1 Protecting Directives

To ensure that your directives are interpreted only by Cray Research compilers, you should use the following coding technique, where *identifier* represents the name of the directive:

```
#if _CRAYC
    #pragma _CRI identifier
#endif
```

This ensures that other compilers used to compile this code will not interpret the directive. Some compilers diagnose any directives that they do not recognize. The Cray Research compilers diagnose directives that are not recognized only if the `_CRI` specification is used.

4.2 Directives in Cray C++

C++ prohibits referencing undeclared objects or functions. Objects and functions must be declared prior to using them in a `#pragma` directive. This is not always the case with C.

Some `#pragma` directives take function names as arguments (for example: `#pragma align`, `#pragma soft`, `#pragma suppress`, `#pragma inline`, and `#pragma noinline`). No overloaded or member functions (no qualified names) are allowed for these directives. This limitation does not apply to the `#pragma` directives for template instantiation.

4.3 Loop Directives

Many directives apply to groups. Unless otherwise noted, these directives must appear before a `for`, `while`, or `do...while` loop. These directives may also appear before a label for `if...goto` loops. If a loop directive appears before a label that is not the top of an `if...goto` loop, it is ignored.

4.4 Alternative Directive Form: _Pragma

Compiler directives can also be specified in the following form, which has the advantage in that it can appear inside macro definitions:

```
_Pragma("_CRI identifier");
```

This form has the same effect as using the #pragma form, except that everything that appeared on the line following the #pragma must now appear inside the double quotation marks and parentheses. The expression inside the parentheses must be a single string literal, but it cannot be a macro that expands into a string literal. _Pragma is a Cray extension to the C and C++ standards.

The following is an example using the #pragma form:

```
#pragma _CRI ivdep
#pragma _CRI parallel private(i, j, k) \
                      shared(a, b, c) \
                      valude(x, y, z)
```

The following is the same example using the alternative form:

```
_Pragma("_CRI ivdep");
_Pragma("_CRI parallel private(i, j, k) \
                       shared(a, b, c)  \
                       value(x, y, z)");
```

Macros are expanded in the string literal argument for _Pragma in an identical fashion to the general specification of a #pragma directive:

```
#define NUM_CHUNKS 8
_Pragma("_CRI parallel numchunks(NUM_CHUNKS)")
```

4.5 General Directives

General directives specify compiler actions that are specific to the directive and have no similarities to the other types of directives. The following sections describe general directives.

4.5.1 besu Directive

The besu directive indicates that *n* BESUs (barrier/eureka synchronization units) should be allocated for use in the compilation unit. The format of this directive is as follows:

```
#pragma _CRI besu n
```

The sum of the BESU counts specified with directives in a program is recorded at link time and placed in the a.out header. The operating system allocates the specified number to the application team at program startup. As a special case, the operating system does not allocate a BESU if the BESU count in the a.out header is 1 and the program used one PE.

For more information on accessing BESUs, see *Barrier and Eureka Synchronization (CRAY T3E Systems)*, publication HMM-141–0. (A nondisclosure agreement must be signed with Cray Research before you can obtain this document.) For a convenient source of BESU state codes, see header file mpp/mpphw_t3e.h.

This directive is not required when accessing BESUs through the following barrier and eureka event routines: barrier(3), pvm_barrier(3), shmem_barrier_all(3), set_event(3), wait_event(3), test_event(3), clear_event(3). However, this directive is required when programming BESUs directly through the techniques described in the *Barrier and Eureka Synchronization (CRAY T3E Systems)*, publication HMM-141–0.

4.5.2 [no]bounds Directive (C Compiler)

The bounds directive specifies that pointer and array references are to be checked. The nobounds directive specifies that this checking is to be disabled.

When bounds checking is in effect, pointer references are checked to ensure that they are not 0 or are not greater than the machine memory limit. Array references are checked to ensure that the array subscript is not less than 0 or greater than or equal to the declared size of the array. Both directives take effect starting with the next program statement in the compilation unit, and stay in effect until the next bounds or nobounds directive, or until the end of the compilation unit.

These directives have the following format:

```
#pragma _CRI bounds

#pragma _CRI nobounds
```

The following example illustrates the use of the bounds directive:

```
int a[30];
#pragma _CRI bounds
void f(void)
{
   int x;
   x = a[30];
   .
   .
   .
}
```

4.5.3 duplicate Directive (C Compiler)

Scope: Global

The `duplicate` directive lets you provide additional, externally visible names for specified functions. You can specify duplicate names for functions by using a directive with one of the following forms:

```
#pragma _CRI duplicate actual as dupname...

#pragma _CRI duplicate actual as (dupname...)
```

The *actual* argument is the name of the actual function to which duplicate names will be assigned. The *dupname* list contains the duplicate names that will be assigned to the actual function. The *dupname* list may be optionally parenthesized. The word, `as`, must appear as shown between the *actual* argument and the comma-separated list of *dupname* arguments.

The `duplicate` directive can appear anywhere in the source file and it must appear in global scope. The actual name specified on the directive line must be defined somewhere in the source as an externally accessible function; the actual function cannot have a static storage class.

Because duplicate names are simply additional names for functions and are not functions themselves, they cannot be declared or defined anywhere in the compilation unit. To avoid aliasing problems, duplicate names may not be referenced anywhere within the source file, including appearances on other directives. In other words, duplicate names may only be referenced from outside the compilation unit in which they are defined.

4.5.4 `message` Directive

The `message` directive directs the compiler to write the message defined by *text* to `stderr` as a warning message. Unlike the `error` directive, the compiler continues after processing a `message` directive. The format of this directive is as follows:

```
#pragma _CRI message "text"
```

4.5.5 `[no]opt` Directive

Scope: Global

The `noopt` directive disables all automatic optimizations and causes optimization directives to be ignored in the source code that follows the directive. Disabling optimization removes various sources of potential confusion in debugging. The `opt` directive restores the state specified on the command line for automatic optimization and directive recognition. These directives have global scope and override related command-line options.

The format of these directives is as follows:

```
#pragma _CRI opt

#pragma _CRI noopt
```

4.5.6 `uses_eregs` Directive

Scope: Local

(UNICOS/mk systems) The `uses_eregs` directive reserves all E registers for your use in the function in which the directive appears. It prevents the compiler from generating code that would change E register values. The format of this directive is as follows:

```
#pragma _CRI uses_eregs
```

The `uses_eregs` directive applies only to the function in which it appears. Your code must comply with E register conventions as described in the *Cray Assembler for MPP (CAM) Reference Manual*.

Note: Use of this directive prevents the cache_bypass directive from being processed because when uses_eregs is in effect, no E registers are available to the compiler.

4.5.7 soft Directive

Scope: Global

The soft directive specifies external identifiers with references that are to be considered soft. *Soft external* references can be to a function or to a data object. Soft externals do not increase your program's total memory requirements.

The format of this directive is as follows:

```
#pragma _CRI soft [var...]
```

var List of one or more soft externals, separated by commas (,) and optionally enclosed in parentheses.

Declaring a soft external directs the linker to link the object or function only if it is already linked (that is, if it has been referenced without soft externals in another code file); otherwise, it is left as an unsatisfied external. If you declare a soft external, you also direct the linker to inhibit an unsatisfied external message if it is left unsatisfied.

Note: The loader treats soft externals as unsatisfied externals, so they remain silently unsatisfied if all references are under the influence of a soft directive. Thus, it is your responsibility to ensure that run-time references to soft external names do not occur unless the loader (using some "hard" reference elsewhere) has actually loaded the entry point in question. You can determine whether a soft external has been loaded by calling the loaded(3) library function.

The soft directive must appear at global scope. Soft externals must have the following attributes:

- They must be declared, but not defined or initialized, in the source file.
- They cannot be declared with a static storage class.
- They cannot be declared as task common.

4.5.8 `vfunction` Directive

Scope: Global

(UNICOS systems) The `vfunction` directive lists external functions that use the call-by-register calling sequence. Such functions can be vectorized but must be written either in Cray Assembly Language (CAL) or in Fortran using the Fortran `vfunction` compiler directive. The format of this directive is as follows:

```
#pragma _CRI vfunction func
```

The *func* variable specifies the name of the external function.

4.5.9 `ident` Directive

The `ident` directive directs the compiler to store the string indicated by *text* into the object (`.o`) file. This can be used to place a source identification string into an object file.

The format of this directive is as follows:

```
#pragma _CRI ident "text"
```

4.6 Instantiation Directives

The Cray C++ compiler recognizes three instantiation directives. Instantiation directives can be used to control the instantiation of specific template entities or sets of template entities.

- The `#pragma _CRI instantiate` directive causes a specified entity to be instantiated.
- The `#pragma _CRI 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 `#pragma _CRI can_instantiate` directive indicates that a 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.

  See Chapter 5, page 91 for more information on template instantiation.

4.7 Vectorization Directives

Because vector operations cannot be expressed directly in Cray Standard C or Cray C++, the compilers must be capable of vectorization, which means transforming scalar operations into equivalent vector operations. The candidates for vectorization are operations in loops and assignments of structures. For more information, see *Optimizing Application Code on UNICOS Systems*.

On UNICOS/mk systems, the compiler can perform "vectorization-like" optimizations on certain loops. Vector versions of the following functions are used when the function appears in a vectorizable loop on UNICOS/mk systems: `alog`(3M), `exp`(3M), `sqrt`(3M), `ranf`(3M), `sin`(3M), `cos`(3M), `coss`(3M), `pow`(3C), and `_popcnt`(3I). This "vectorization" is performed using the following process:

1. The loop is stripmined. Stripmining is a single-processor optimization technique in which arrays and the program loops that reference them are split into optimally-sized blocks, termed strips. The original loop is transformed into two nested loops. The inner loop references all data elements within a single strip, and the outer loop selects the strip to be addressed in the inner loop. This technique is often performed by the compiler to maximize the usage of cache memory or as part of vector code generation.

2. If necessary, a strip of operands is stored in a temporary array. The vector version of the function is called, which stores the strip of results in a temporary array.

3. The remainder of the loop is computed using the results from step 2.

The subsections that follow describe the compiler directives used to control vectorization on UNICOS systems and "vectorization-like" optimizations on UNICOS/mk systems.

4.7.1 `ivdep` Directive

Scope: Local

The `ivdep` directive tells the compiler to ignore vector dependencies for the loop immediately following the directive. Conditions other than vector dependencies can inhibit vectorization. If these conditions are satisfactory, the loop vectorizes. This directive is useful for some loops that contain pointers and indirect addressing. The format of this directive is as follows:

```
#pragma _CRI ivdep
```

4.7.2 novector Directive

Scope: Local

The novector directive directs the compiler to not vectorize the loop that immediately follows the directive. It overrides any other vectorization-related directives, as well as the -h vector and -h ivdep command-line options. The format of this directive is as follows:

```
#pragma _CRI novector
```

4.7.3 novsearch Directive

Scope: Local

(UNICOS systems) The novsearch directive directs the compiler to not vectorize the search loop that immediately follows the directive. A search loop is a loop with one or more early exit statements. It overrides any other vectorization-related directives as well as the -h vector and -h ivdep command-line options. The format of this directive is as follows:

```
#pragma _CRI novsearch
```

4.7.4 prefervector Directive

Scope: Local

(UNICOS systems) The prefervector directive tells the compiler to vectorize the loop that immediately follows the directive if the loop contains more than one loop in the nest that can be vectorized. The directive states a vectorization preference and does not guarantee that the loop has no memory dependence hazard.

The format of this directive is as follows:

```
#pragma _CRI prefervector
```

4.7.5 shortloop and shortloop128 Directives

Scope: Local

The shortloop (all systems) and shortloop128 (UNICOS systems only) directives improve performance of a vectorized loop by allowing the compiler to omit the run-time test to determine whether it has been completed. The shortloop compiler directive identifies vector loops that execute with a maximum iteration count of 64 (504 for character arrays) and a minimum iteration count of 1. The shortloop128 compiler directive identifies vector loops that execute with a maximum iteration count of 128 (1016 for character arrays) and a minimum iteration count of 1. If the iteration count is outside the range for the directive, results are unpredictable.

These directives are ignored if the loop trip count is known at compile time and is greater than the target machine's vector length. The vector length of Cray C90 systems and Cray T90 systems is 128. The vector length of all other UNICOS systems is 64.

The formats of these directives are as follows:

```
#pragma _CRI shortloop
#pragma _CRI shortloop128
```

4.8 Tasking Directives

The Cray Standard C compiler and Cray C++ compiler support parallel processing using multiple processors on UNICOS systems. Parallel processing is a technique that breaks a computational task into a set of subtasks and then performs each subtask simultaneously. This allows many jobs to run faster by spreading a computational task across multiple processors. The increase in speed of execution depends on the degree of parallelism that is inherent in the program. See *Optimizing Application Code on UNICOS Systems*, for more information.

Tasking can be performed automatically by the compiler (Autotasking) or it can be directed by the user (user-directed tasking). The methods that can be used to accomplish tasking are defined as follows:

- Autotasking is performed automatically by the compiler based on its analysis of the code.

  Autotasking is enabled by specifying the -h task*n* option on the command line. For more information on the -h task*n* option, see Section 2.11.1, page 16.

- *User-directed tasking*, sometimes called *microtasking* or simply *tasking*, is controlled by the directives you add to your code. This requires that you understand the requirements for tasking and perform your own analysis.

  To direct tasking manually, you must identify the regions of your program that are to run in parallel; then insert tasking directives to specify these regions to the compiler.

4.8.1 `parallel` and `endparallel` Directives

Scope: Local

(UNICOS systems) The `parallel` directive marks the start of a parallel region. The `endparallel` directive marks the end of a parallel region. Parallel regions are combinations of redundant code blocks (executed by all processors) and partitioned code blocks (portions executed by each processor, such as the iterations of a tasked loop). The `parallel` directive indicates where multiple processors enter execution, which may be different from where they demonstrate a direct benefit (partitioned code block). The format of these directives is as follows:

```
#pragma _CRI parallel [shared(var...)] [private(var...)]
   [value(var...)] [defaults] [if (exp)] [maxcpus (exp)]

#pragma _CRI endparallel
```

Arguments to tasking directives are described in Section 4.8.11, page 78.

4.8.2 `taskloop` Directive

Scope: Local

(UNICOS systems) The `taskloop` directive indicates that the following `for` loop can be executed in parallel by multiple processors. Although no directive is needed to end a `taskloop` loop, the `endloop` directive (see Section 4.8.3, page 73) can be used to explicitly do so. Unlike other loop-based directives, the `taskloop` directive must appear before a `for` loop.

The `taskloop` directive can be used either inside or outside of a parallel region. When the directive is used inside a parallel region, the `private`, `shared`, `value`, `defaults`, `if`, and `maxcpus` arguments are not allowed. These arguments, if specified, must be specified on the `parallel` directive that precedes the `taskloop` directive. When a `taskloop` directive is used outside

a parallel region, the loop is referred to as a ***stand-alone task loop***. The savelast argument can be specified only on stand-alone task loops.

For task loops outside of a parallel region, the format of the taskloop directive is as follows:

```
#pragma _CRI taskloop [shared(var)...)] [private(var...)]
  [value(var...)] [defaults] [if (exp)] [maxcpus(exp)]
  [savelast] [dist]
```

For task loops inside a parallel region, the format of the taskloop directive is as follows:

```
#pragma _CRI taskloop [dist]
```

Arguments to tasking directives are described in Section 4.8.11, page 78.

4.8.3 endloop Directive

Scope: Local

(UNICOS systems) By default, a directive is not needed to end a taskloop loop. The endloop directive is a special terminator for the taskloop directive inside a parallel region. The endloop directive extends the range of the control structure that contains the taskloop loop. This allows a mechanism to exploit parallelism in loops that contain reduction computations. The endloop directive can appear only in a parallel region. The format of the endloop directive is as follows:

```
#pragma _CRI endloop
```

4.8.4 case and endcase Directives

Scope: Local

(UNICOS systems) The case directive serves as a separator between adjacent code blocks that are concurrently executable. The case directive can appear only in a parallel region. The endcase directive serves as the terminator for a group of one or more parallel cases.

The format of the case and endcase directives is as follows:

```
#pragma _CRI case

#pragma _CRI endcase
```

4.8.5 guard and endguard Directives

Scope: Local

(UNICOS systems) The guard and endguard directive pair delimit a guarded region and provide the necessary synchronization to protect (or guard) the code inside the guarded region. A *guarded region* is a code block that is to be executed by only one processor at a time, although all processors in the parallel region execute it.

The format of the guard and endguard directives is as follows:

```
#pragma _CRI guard [exp]

#pragma _CRI endguard [exp]
```

Unnumbered guards do not use the optional parameter *exp* on the guard and endguard directives. Only one processor is allowed to execute in an unnumbered guarded region at a time. If a processor is executing in an unnumbered guarded region, and a second processor wants to enter an unnumbered guarded region, the second processor must wait until the first processor exits the region.

Numbered guards are indicated by the use of the optional parameter *exp*. The expression *exp* must be an integral expression. Only the low-order 6 bits of *exp* are used, thereby allowing up to 64 distinct numbered guards (0 through 63). For optimal performance, *exp* should be an integer constant; the general expression capability is provided only for the unusual case that the guarded region number must be passed to a lower-level function.

4.8.6 taskprivate Directive (C Compiler)

The taskprivate directive specifies the task private storage class for variables. The format of this directive is as follows (the comma-separated list of variables can be enclosed in parentheses):

```
#pragma _CRI taskprivate variable,...
```

Variables that are given a task private storage class are placed in storage so that each task has a separate copy of the variables; all functions within a task can access the same copy of the task private variable, but no task can access any task private variables belonging to another task.

A primary use for task private variables is efficient porting of macrotasked programs from a shared-memory system (that is, a system, such as VAX, on which independently executing programs can access the other program's memory). On Cray systems, independently executing programs cannot access memory belonging to other programs.

This directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with `static` storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as task private, subsequent declarations of that variable in the same source file inherit the task private storage class.

The `taskprivate` directive takes precedence over the `-h common` and the `-h taskcommon` command-line options.

The following restrictions apply to the `taskprivate` variable:

- A `taskprivate` variable cannot also be a soft external.
- The address of a `taskprivate` variable cannot be taken in a constant expression (for example, an initializer).

4.8.7 `taskshared` Directive (C Compiler)

The `taskshared` directive ensures that specified variables are accessible to all tasks (not stored as task private). For example, you can use this directive with the `-h taskprivate` option, to exempt certain variables that would otherwise be task private. The `taskshared` directive overrides the `-h taskprivate` and `-h taskcommon` command-line options.

The format of this directive is as follows (the comma-separated list of variables can be placed in parentheses):

```
#pragma _CRI taskshared variable,...
```

The taskshared directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with static storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as task shared, subsequent declarations of that variable in the same source file inherit the task shared storage class.

4.8.8 taskcommon Directive

The taskcommon directive specifies the task common storage class for variables. The format of this directive is as follows (the comma-separated list of variables can be placed in parentheses):

```
#pragma _CRI taskcommon variable, ...
```

Variables that are given a task common storage class are placed in storage so that each task has a separate copy of the variables; all functions within a task can access the same copy of the task common variable, but no task can access any task common variables belonging to another task.

A primary use for task common variables is efficient porting of macrotasked programs from a shared-memory system (that is, a system, such as VAX, on which independently executing programs can access the other program's memory). On Cray systems, independently executing programs cannot access memory belonging to other programs.

This directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with static storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as task common, subsequent declarations of that variable in the same source file inherit the task common storage class.

The taskcommon directive takes precedence over the -h common and -h taskprivate command-line options.

The following restrictions apply to taskcommon variables:

- A taskcommon variable cannot be initialized. (A taskprivate variable can be initialized, see Section 4.8.6, page 74.) By default, a taskcommon variable is initialized to 0.
- A taskcommon variable cannot also be a soft external.
- The address of a taskcommon variable cannot be taken in a constant expression (for example, an initializer).

4.8.9 common Directive

A common directive ensures that specified variables are accessible to all tasks (not stored as taskcommon). Use this directive, for example, with the -h taskcommon option, to exempt certain variables that would otherwise be taskcommon. The common directive overrides the -h taskcommon and -h taskprivate command-line options. The format of the common directive is as follows (the comma-separated list of variables can be placed in parentheses):

```
#pragma _CRI common variable,...
```

The common directive can appear in both global and local scopes and applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Local scope variables with static storage class, in which case the directive must appear within the same scope as the variable declaration.

When a variable is designated as common, subsequent declarations of that variable in the same source file inherit the common storage class.

4.8.10 prefertask Directive

Scope: Local

(UNICOS systems) The prefertask directive tells the compiler to generate tasked code for the loop that immediately follows it if that loop contains more

than one loop in the nest that can be tasked. The directive states a tasking preference but does not guarantee that the loop has no memory dependence hazard. Aggressive tasking (enabled by the `-h task3` command-line option) must be enabled for this directive to take effect. Threshold testing for the loop specified by using the `prefertask` directive is suppressed. The format of the `prefertask` directive is as follows:

```
#pragma _CRI prefertask
```

4.8.11 Arguments to Tasking Directives

The tasking directive arguments are categorized as context arguments, work distribution arguments, or miscellaneous arguments. Arguments can appear in any order in the directive. See the *Cray C/C++ Reference Manual*, for a detailed description of these arguments.

4.8.11.1 Context Arguments

The following tasking directive arguments are used to indicate the tasking context for variables referenced in the parallel region. Tasking context is an attribute that determines how the different processors access a variable in a parallel region.

If the `private`, `shared`, or `value` argument is used, at least one variable must be declared in the corresponding list. For these arguments, variable names follow the argument in a comma-separated list enclosed in parentheses.

- `private`
- `shared`
- `value`
- `defaults`

4.8.11.2 Work Distribution Arguments

The following arguments specify the work distribution policy for the iterations of a tasked loop and can be used only with the `taskloop` directive. By default, the iterations are handed out to the available processors in parallel, one iteration per processor, one at a time (this is the `single` work distribution). Only one of the following work distribution arguments can be specified for a given loop.

For all work distribution arguments except `single`, each chunk of iterations can be vectorized, subject to the normal rules for vectorization.

- `single`
- `chunksize (`***exp***`)`
- `numchunks (`***exp***`)`
- `guided`
- `vector`

4.8.11.3 Miscellaneous Arguments

The following arguments specify miscellaneous arguments that do not fit either of the previous categories.

- `if (`***exp***`)`
- `maxcpus (`***exp***`)`
- `savelast`

4.9 Multi-Streaming Processor (MSP) Directives (Deferred Implementation)

The following section describes `#pragma` directives for the Multi-Streaming Processor (MSP) optimization.

> **Note:** MSP is an optional feature. To determine whether MSP is enabled on your system, enter the `sysconf` command at your system prompt. The `HARDWARE` output field contains the `NMSP=` field that shows the number of MSPs configured. For more information, see the `sysconf`(1) man page.

(Cray SV1 systems) The MSP directives work with the `-h stream`***n*** command line option to determine whether parts of your program are optimized for the MSP. One of the following options must be specified on the C or C++ command line in order for these directives to be recognized: `-h stream1`, `-h stream2`, or `-h stream3`. For more information on the `-h stream`***n*** command line option, see Section 2.9.1, page 14.

The MSP `#pragma` directives are as follows:

- `#pragma preferstream`
- `#pragma nostream`

The following sections describe the MSP optimization directives.

4.9.1 `#pragma nostream` Directive

Scope: Local

(Cray SV1 Systems) The `#pragma nostream` directive directs the compiler to not perform MSP optimizations on the loop that immediately follows the directive. It overrides any other MSP-related directives as well as the `-h stream`*n* command-line option.

The format of this directive is as follows:

```
#pragma _CRI nostream
```

The following example illustrates the use of the `#pragma nostream` directive:

```
#pragma _CRI nostream
   for ( i = 0; i < n1; i++ ) {
         x[i] = y[i] + z[i]
   }
```

4.9.2 `#pragma preferstream` Directive

Scope: Local

(Cray SV1 Systems) For cases in which the compiler could perform MSP optimizations on more than one loop in a loop nest, the `#pragma preferstream` directive tells the compiler that the loop immediately following the directive is the one to be optimized.

The format of this directive is as follows:

```
#pragma _CRI preferstream
```

The following example illustrates the use of the #pragma preferstream directive:

```
   for ( j = 0; j< n2; j++ ) {
#pragma _CRI preferstream
      for ( i = 0; i < n1; i++ ) {
         a[j][i] = b[j][i] + c[j][i]
      }
   }
```

4.10 Scalar Directives

The following subsections describe the scalar optimization directives, which control aspects of code generation, register storage, and so on.

4.10.1 align Directive

(UNICOS systems) The align directive causes functions, loops, or labels to be aligned on instruction buffer boundaries. This increases the size of the compiled program but improves loop performance. When used in global scope, the align directive specifies that functions be aligned on instruction buffer boundaries. When used in local scope, this directive lets you specify that the loop or label following the directive is to be aligned on an instruction buffer boundary.

To determine the number and size of the instruction buffers on your system, use the target(1) command described in the *UNICOS User Commands Reference Manual*.

4.10.2 cache_align Directive

Systems:

(UNICOS/mk systems) The cache_align directive aligns each specified variable on a cache line boundary. This is useful for frequently referenced variables. A *cache* is storage that can be accessed more quickly than conventional memory. A *cache line* is a division within a cache. Properly used, the cache_align directive lets you prevent cache conflicts.

The directive's effect is independent of its position in source. It can appear in global or local scope. The format of the cache_align directive is as follows:

```
#pragma _CRI cache_align var_list
```

In the preceding format, *var_list* represents a list of variable names separated by commas.

4.10.3 cache_bypass Directive

Scope: Local

(UNICOS/mk systems) The `cache_bypass` directive specifies that local memory references in a loop should be passed through E registers.

E registers offer fine-grained access to local memory and a higher bandwidth for sparse index array accesses such as gather/scatter operations and large-stride accesses. These operations do not exploit the spatial locality of cache references. Using this directive can greatly decrease run time for gather/scatter operations. The benefits of using this directive are higher with random index streams. Using this directive increases the latency of memory references in return for greater bandwidth, so this directive may increase runtime for loops with a high degree of spatial locality that derive benefit from cache references.

E registers can also be used to initialize large arrays that contain data not immediately needed in cache. This avoids unnecessary reads into cache and improves memory bandwidth efficiency for the initialization.

The format of the `cache_bypass` directive is as follows:

```
#pragma _CRI cache_bypass var,...
```

var, . . . One or more variable names. The variable must have type array of or pointer to (array of or pointer to...) a 64–bit scalar type.

This directive precedes the loop that contains data to be accessed through E registers. If both a `cache_bypass` and a `novector` directive are applied to the same loop, the `novector` directive is ignored,

The compiler ignores the `cache_bypass` directive if it determines that it cannot generate code efficiently. To increase the probability of this directive being used, the loop should have the following characteristics:

- The loop must be an inner loop (it must not contain other loops).
- The loop must be vectorizable. You may need to use the `ivdep` directive in conjunction with `cache_bypass` to ensure that the loop is processed.

- The base array or pointer within the loop must be invariant.

To see the most benefit from the cache_bypass directive, you may want to enable loop unrolling. For information on the command-line option to control unrolling, see Chapter 2, page 3.

This feature may disable the UNICOS/mk system stream buffer hardware feature for the entire program. This is done on certain Cray T3E platforms because the compiler cannot guarantee correctness in terms of the interaction of the stream buffers and the E register operations generated by this directive. Disabling stream buffers can cause considerable performance degradation for other parts of your program. The stream buffer features can be reenabled by using the set_d_stream(3) library function. Consult with your system administrator to determine whether your Cray T3E system falls into this category. If so, see the streams_guide(7) man page for details on how and when streams can be safely reenabled in the presence of E register operations.

4.10.4 concurrent Directive

Scope: Local

The concurrent directive indicates that no dependencies exist between different array references. When the compiler cannot disambiguate between different array references, it is assumed that a dependency exists (for safety), when none may exist.

Specifying the optional safe_distance=*n* argument indicates that no dependencies exist between the current iteration of the loop and *n* subsequent iterations. *n* must be an integral constant greater than zero.

The format of the concurrent directive is as follows:

```
#pragma _CRI concurrent [safe_distance=n]
```

This directive should immediately precede the loop that will benefit from the directive.

4.10.5 nointerchange Directive

Scope: Local

The nointerchange directive inhibits the compiler's ability to interchange the loop that follows the directive with another inner or outer loop.

The format of this directive is as follows:

```
#pragma _CRI nointerchange
```

4.10.6 `noreduction` Directive

Scope: Local

The `noreduction` compiler directive tells the compiler to not optimize the loop that immediately follows the directive as a reduction loop. If the loop is not a reduction loop, the directive is ignored.

A ***reduction loop*** is a loop that contains at least one statement that reduces an array to a scalar value by doing a cumulative operation on many of the array elements. This involves including the result of the previous iteration in the expression of the current iteration.

You may choose to use this directive when the loop iteration count is small or when the order of evaluation is numerically significant. It overrides any vectorization-related directives as well as the `-h vector` and `-h ivdep` command-line options. The effect of the `noreduction` directive differs depending on your platform.

On UNICOS systems, the `noreduction` directive disables vectorization of any loop that contains a reduction. The specific reductions that are disabled are summation and product reductions, and alternating value computations. On UNICOS/mk systems, the `noreduction` directive prevents the compiler from rewriting loops involving multiplication or exponentiation by an induction variable to be a series of additions or multiplications of a value.

Regardless of platform, however, the format of this directive is as follows:

```
#pragma _CRI noreduction
```

4.10.7 `split` Directive

Scope: Local

(UNICOS/mk systems) The `split` directive instructs the compiler to attempt to split the following loop into a set of smaller loops.

Such loop splitting attempts to improve single processor performance by making best use of the six stream buffers of the UNICOS/mk system. It achieves this by splitting an inner loop into a set of smaller loops, each of which allocates no

more than six stream buffers, thus avoiding stream buffer thrashing. The stream buffer feature reduces memory latency and increases memory bandwidth by prefetching for long, small-strided sequences of memory references.

The `split` directive has the following format:

```
#pragma _CRI split
```

The `split` directive merely asserts that the loop can profit by splitting. It will not cause incorrect code.

The compiler splits the loop only if it is safe. Generally, a loop is safe to split under the same conditions that a loop is vectorizable. The compiler only splits inner loops. The compiler may not split some loops with conditional code.

The `split` directive also causes the original loop to be stripmined. This is done to increase the potential for cache hits between the resultant smaller loops.

Loop splitting can reduce the execution time of a loop by as much as 40%. Candidates for loop splitting can have trip counts as low as 40. They must also contain more than six different memory references with strides less than 16.

Note that there is a slight potential for increasing the execution time of certain loops. Loop splitting also increases compile time, especially when loop unrolling is also enabled.

If both a `split` and a `novector` directive are applied to the same loop, the `novector` directive is ignored.

4.10.8 `suppress` Directive

The `suppress` directive suppresses optimization in two ways, determined by its use with either global or local scope.

The global scope `suppress` directive specifies that all associated local and task common variables are to be written to memory before a call to the specified function. This ensures that the value of the variables will always be current. The global `suppress` directive takes the following form:

```
#pragma _CRI suppress func...
```

The local scope `suppress` directive stores current values of the specified variables in memory. If the directive lists no variables, all variables are stored to

memory. This directive causes the values of these variables to be reloaded from memory at the first reference following the directive. The local suppress directive has the following format:

```
#pragma _CRI suppress [var...]
```

The net effect of the local suppress directive is similar to declaring the affected variables to be volatile except that the volatile qualifier affects the entire program whereas the local suppress directive affects only the block of code in which it resides.

On UNICOS/mk systems, suppress, with no arguments specified, invalidates the entire cache or forces all entities in the cache to be read from memory. This gives suppress a higher performance cost than it has on other architectures, so specifying particular variables can be more efficient.

4.10.9 symmetric Directive

Scope: Local

(UNICOS/mk systems) The symmetric directive declares that an auto or register variable has the same local address on all processing elements (PEs). This is useful for global addressing using the shmem library functions. For information on the shmem library functions, see the intro_shmem(3) man page. The format for this compiler directive is as follows:

```
#pragma _CRI symmetric var...
```

The symmetric directive must appear in local scope. Each variable listed on the directive must:

- Be declared in the same scope as the directive.
- Have auto or register storage class.
- Not be a function parameter.

Because all PEs must participate in the allocation of symmetric stack variables, there is an implicit barrier before the first executable statement in a block containing symmetric variables.

If a goto statement jumps into a block where a symmetric variable has been declared, the behavior is undefined. If a block is exited by means of a goto,

longjmp, and so on, the memory associated with any symmetric variables declared in that block will be squandered. Neither of these conditions are detected by the compiler.

4.10.10 unroll Directive

Scope: Local

The unrolling directive allows the user to control unrolling for individual loops.

Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- Improved loop scheduling by increasing basic block size
- Reduced loop overhead
- Improved chances for cache hits

The format for this compiler directive is as follows:

```
#pragma _CRI unroll [n]
```

The *n* argument specifies the total number of loop body copies to be generated. *n* must be in the range of 2 through 63.

If you do not specify a value for *n*, the compiler attempts to determine the number of copies to generate based on the number of statements in the loop nest.

Caution: If placed prior to a noninnermost loop, the unroll directive asserts that the following loop has no dependencies across iterations of that loop. If dependencies exist, incorrect code could be generated.

The unroll compiler directive can be used only on loops with iteration counts that can be calculated before entering the loop. If unroll is specified on a loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only one loop, and the innermost loop can contain work.

The compiler can be directed to attempt to unroll all loops generated for the program with the -h unroll command-line option.

On UNICOS/mk systems, the amount of unrolling specified on the `unroll` directive overrides those chosen by the compiler when the `-h unroll` command-line option is specified. On UNICOS systems, the compiler may do additional unrolling over the amount requested by the user.

Outer loop unrolling is not always legal because the transformation can change the semantics of the original program.

4.11 Inlining Directives

Inlining replaces calls to user-defined functions with the code representing the function. This can improve performance by saving the expense of the function call overhead. It also enhances the possibility of additional code optimization and vectorization, especially if the function call was an inhibiting factor.

Inlining is invoked in the following ways:

- Automatic inlining of an entire compilation is enabled by issuing the `-h inline` command-line option, as described in Section 2.12.1, page 18.
- Inlining of particular function calls is specified by the `inline` directive, as discussed in the following sections.

Inlining directives can appear in global scope (that is, not inside a function definition). Global inlining directives specify whether all calls to the specified functions should be inlined (`inline` or `noinline`).

Inlining directives can also appear in local scope; that is, inside a function definition. A local inlining directive applies only to the next call to the function specified on the directive. Although the function specified on an inlining directive does not need to appear in the next statement, a call to the function must occur before the end of the function definition.

Inlining directives always take precedence over the automatic inlining requested on the command line. This means that function calls that are associated with inlining directives are inlined before any function calls selected to be inlined by automatic inlining.

The `-h report=i` option writes messages identifying where functions are inlined or briefly explains why functions are not inlined.

4.11.1 `inline` Directive

The `inline` directive specifies functions that are to be inlined. The `inline` directive has the following format:

```
#pragma _CRI inline func,...
```

The *func,...* argument represents the function or functions to be inlined. The list can be enclosed in parentheses. Listed functions must be defined in the compilation unit. You cannot specify objects of type pointer-to-function.

4.11.2 `noinline` Directive

The `noinline` directive specifies functions that are not to be inlined. The format of the `noinline` directive is as follows:

```
#pragma _CRI noinline func,...
```

The *func,...* argument represents the function or functions that are not to be inlined. The list can be enclosed in parentheses. Listed functions must be defined in the compilation unit. You cannot specify objects of type pointer-to-function.

Template Instantiation [5]

A *template* describes a class or function that is a model for a family of related classes or functions. The act of generating a class or function from a template is called *template instantiation*.

For example, a template can be created for a stack class, and then a stack of integers, a stack of floats, and a stack of some user-defined type can be used. In source code, these might be written as `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed during a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (template entities) are not necessarily done immediately for the following reasons:

- The preferred end result is one copy of each instantiated entity across all object files in a program. This applies to entities with external linkage.
- A specialization of a template entity is allowed. For example, a specific version of `Stack<int>`, or of just `Stack<int>::push` could be written to replace the template-generated version and to provide a more efficient representation for a particular data type.

  Because the compiler does not know about specializations of entities provided in future compilations when compiling a reference to a template entity, it cannot automatically instantiate the template in source files that contain references to the template.
- If a template function is not referenced, it should not be compiled because such functions could contain semantic errors that would prevent compilation. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

Note: Certain template entities, such as inline functions, are always instantiated when they are used.

If the compiler is responsible for doing all instantiations automatically, it can only do so for the entire program. That is, the compiler cannot make decisions about instantiation of template entities until all source files of the complete program have been read.

The Cray C++ compiler provides an instantiation mechanism that does automatic instantiation at linkage and provides command-line options and `#pragma` directives that give the programmer more explicit control over instantiation.

5.1 Automatic Instantiation

The goal of an automatic instantiation mode is to provide trouble-free instantiation. The programmer should be able to compile source files to object code, link them and run the resulting program, without questioning how the necessary instantiations are done.

In practice, this is difficult for a compiler to do, and different compilers use different automatic instantiation schemes with different strengths and weaknesses.

The Cray C++ compiler requires a normal, top-level, explicitly compiled source file that contains the definition of both the template entity and of any types required for the particular instantiation. This requirement is met in one of the following ways:

- Each `.h` file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- When the compiler sees a template declaration in a .h file and discovers a need to instantiate that entity, implicit inclusion gives the compiler permission to search for an associated definition file having the same base name and a different suffix and implicitly include that file at the end of the compilation (see Section 5.4, page 97).
- The programmer makes sure that the files that define template entities also have the definitions of all the available types and adds code or directives in those files to request instantiation of those entities.

Automatic instantiation is accomplished by the Cray C++ compiler as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation. For any source file that makes use of a template instantiation, an associated `.ii` file is created, if one does not already exist (for example, the compilation of `abc.C` results in the creation of `abc.ii`).
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 any additional information about entities that could be instantiated.

Caution: The prelinker does not examine the object files in a library (.a) file.

3. If the prelinker finds a reference to a template entity for which there is no definition in the set of object files, it looks for a file that indicates that it could instantiate that template entity. Upon discovery of such a file, it assigns the instantiation to that file. The set of instantiations assigned to a given file (for example, abc.C) is recorded in an associated file that has a .ii suffix (for example, abc.ii).

4. The prelinker then executes the compiler to again recompile each file for which the .ii was changed.

5. During compilation, the compiler obeys the instantiation requests contained in the associated .ii file and produces a new object file that contains the requested template entities and the other things that were already in the object file.

6. The prelinker repeats steps 3 through 5 until there are no more instantiations to be adjusted.

7. The object files are linked together.

Once the program has been linked correctly, the .ii files contain a complete set of instantiation assignments. If source files are recompiled, the compiler consults the .ii files and does the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done. This is true even if the entire program is recompiled. Because the .ii file contains information on how to recompile when instantiating, it is important that the .o and .ii files are not moved between the first compilation and linkage.

The prelinker cannot instantiate into and from library files (.a), so if a library is to be shared by many applications its templates should be expanded. You may find that creating a directory of objects with corresponding .ii files and the use of -h prelink_copy_if_nonlocal (see Section 2.5.6, page 9) will work as if you created a library (.a) that is shared.

The -h prelink_local_copy option indicates that only local files (for example, files in the current directory) are candidates for assignment of instantiations. This option is useful when you are sharing some common

relocatables but do not want them updated. Another way to ensure that shared `.o` files are not updated is to use the `-h remove_instantiation_flags` option when compiling the shared `.o` files. This also makes smaller resulting shared `.o` files.

An easy way to create a library that instantiates all references of templates within the library is to create an empty `main` function and link it with the library, as shown in the following example. The prelinker will instantiate those template references that are within the library to one of the relocatables without generating duplicates. The empty `dummy_main.o` file is removed prior to creating the `.a` file.

```
CC a.C b.C c.C dummy_main.C
bld -q mylib.a a.o b.o c.o
```

If a specialization of a template entity is provided somewhere in the program, the specialization is seen as a definition by the prelinker. Because that definition satisfies the references to that entity, the prelinker will not request an instantiation of the entity. If a specialization of a template is added to a previously compiled program, the prelinker removes the assignment of the instantiation from the proper `.ii` file.

The `.ii` files do not, in general, require any manual intervention. The exception occurs when a definition is changed in such a way that some instantiation no longer compiles (it receives errors) and at the same time a specialization is added to another file and the first file is recompiled before the specialization file. If this exception occurs, the `.ii` file that corresponds to the file that generated the errors must be deleted manually to allow the prelinker to regenerate it.

Automatic instantiation can coexist with partial explicit control of instantiation by the programmer through the use of `#pragma` directives or the `-h instantiate=`***mode*** option.

Automatic instantiation mode can be disabled by issuing the `-h noautoinstantiate` command-line option. If automatic instantiation is disabled, the information about template entities that could be instantiated in a file is not included in the object file.

5.2 Instantiation Modes

Normally, during compilation of a source file, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed by issuing the

-h instantiate=***mode*** command-line option. The ***mode*** argument can be specified as follows:

mode	Description
none	Do not automatically create instantiations of any template entities. This is the most appropriate mode when automatic instantiation is enabled. This is the default instantiation mode.
used	Instantiate those template entities that were used in the compilation. This includes all static data members that have template definitions.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members are instantiated, regardless of whether they were used. Nonmember template functions are instantiated even if the only reference was a declaration.
local	Similar to used mode, except that the functions are given internal linkage. This mode provides a simple mechanism for those who are not familiar with templates. The compiler instantiates the functions used in each compilation unit as local functions, and the program links and runs correctly (barring problems due to multiple copies of local static variables). This mode may generate multiple copies of the instantiated functions and is not suitable for production use. This mode cannot be used in conjunction with automatic template instantiation. Automatic instantiation is disabled by this mode.

In the case where the CC(1) command is given a single source file to compile and link, all instantiations are done in the single source file and, by default, the used mode is used and automatic instantiation is suppressed.

5.3 Instantiation #pragma Directives

Instantiation #pragma directives can be used in source code to control the instantiation of specific template entities or sets of template entities. There are three instantiation #pragma directives:

- The #pragma _CRI instantiate directive causes a specified entity to be instantiated.

- The #pragma _CRI 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 #pragma _CRI can_instantiate directive indicates that a 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 argument to the #pragma _CRI instantiate directive can be any of the following:

- A template class name. For example: A<int>
- A template class declaration. For example: class A<int>
- A member function name. For example: A<int>::f
- A static data member name. For example: A<int>::i
- A static data declaration. For example: int A<int>::i
- A member function declaration. For example: void A<int>::f(int, char)
- A template function declaration. For example: char* f(int, float)

A #pragma directive in which the argument is a template class name (for example, A<int> or class A<int>) is equivalent to repeating the directive for each member function and static data member declared in the class. When instantiating an entire class, a given member function or static data member may be excluded using the #pragma _CRI do_not_instantiate directive. For example:

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

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the #pragma instantiate directive and no template definition is available or a specific definition is provided, an error is issued.

The following example illustrates the use of the #pragma _CRI instantiate directive:

```
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {}  // Specific definition
void main()
{
 int    i;
 double  d;
 f1(i);
 f1(d);
 g1(i);
 g1(d);
}
#pragma _CRI instantiate void f1(int) // error-specific definition
#pragma _CRI instantiate void g1(int) // error-no body provided
```

In the preceding example, `f1(double)` and `g1(double)` are not instantiated because no bodies are supplied, but no errors will be produced during the compilation. If no bodies are supplied at link time, a linker error is issued.

A member function name (such as `A<int>::f`) can be used as a `#pragma` directive argument 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 in the following example:

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

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

5.4 Implicit Inclusion

The implicit inclusion feature implies that if the compiler needs a definition to instantiate 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 will look to see if a file `xyz.C` exists and, if so, it processes 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 Cray C++ compiler must know the full path name to the file in which the template was declared and whether the file was included using the system include syntax (such as `#include <file.h>`). This information is not available for preprocessed source code containing `#line` directives. Consequently, the Cray C++ compiler does not attempt implicit inclusion for source code that contains `#line` directives.

The set of definition-file suffixes that are tried by default, is `.c`, `.C`, `.cpp`, `.CPP`, `.cxx`, `.CXX`, and `.cc`.

Implicit inclusion works well with automatic instantiation, however, they are independent. They can be enabled or disabled independently, and implicit inclusion is still useful without automatic instantiation.

Predefined Macros [6]

Predefined macros can be divided into the following categories:

- Macros required by the C and C++ standards
- Macros based on the host machine
- Macros based on the target machine
- Macros based on the compiler

Predefined macros provide information about the compilation environment. In the subsections that follow, only those macros that begin with the underscore (_) character are defined when running in strict-conformance mode (see the `-h conform` command-line option in Section 2.4.1, page 6).

> **Note:** Any of the predefined macros except those required by the standard (see Section 6.1, page 99) can be undefined by using the `-U` option; they can also be redefined by using the `-D` option.

A large set of macros is also defined in the standard header files. These macros are described in the *UNICOS System Libraries Reference Manual*.

6.1 Macros Required by the C and C++ Standards

The following macros are required by the C and C++ standards:

Macro	Description
`__TIME__`	Time of translation of the source file.
`__DATE__`	Date of translation of the source file.
`__LINE__`	Line number of the current line in your source file.
`__FILE__`	Name of the source file being compiled.
`__STDC__`	Defined as the decimal constant 1 if compilation is in strict conformance mode; defined as the decimal constant 2 if the compilation is in

	extended mode. This macro is defined for C and C++ compilations.
`__cplusplus`	Defined as 1 when compiling C++ code and undefined when compiling C code. The `__cplusplus` macro is required by the ISO C++ draft standard, but not the ISO C standard.

6.2 Macros Based on the Host Machine

The following macros provide information about the environment running on the host machine:

Macro	Description
`__unix`	Defined as 1 if the operating system is UNIX.
`unix`	Defined as 1 if the operating system is UNIX. This macro is not defined in strict-conformance mode.
`_UNICOS`	Defined as the integer portion of the major release level of the current UNICOS release (for example, 9).

6.3 Macros Based on the Target Machine

The following macros provide information about the characteristics of the target machine:

Macro	Description
`cray`	Defined as 1 on all Cray Research systems. This macro is not defined in strict-conformance mode.
`CRAY`	Defined as 1 on all Cray Research systems. This macro is not defined in strict-conformance mode.
`_CRAY`	Defined as 1 on all Cray Research systems.
`CRAY1`	Defined as 1 on all UNICOS systems; if the hardware is any other machine type, the macro is

	not defined. This macro is not defined in strict-conformance mode.
`_CRAY1`	Defined as 1 on all UNICOS systems; if the hardware is any other machine type, the macro is not defined.
`_CRAYMPP`	Defined as 1 on all Cray MPP systems (UNICOS/mk systems); if the hardware is any other machine type, the macro is not defined.
`_CRAYT3E`	Defined as 1 on Cray T3E systems; if the hardware is any other machine type, the macro is not defined.
`_CRAYIEEE`	Defined as 1 if the targeted CPU type uses IEEE floating-point format; if Cray format is used, the macro is not defined.
`_ADDR32`	Defined as 1 if the targeted CPU has 32-bit address registers; if the targeted CPU does not have 32–bit address registers, the macro is not defined.
`_ADDR64`	Defined as 1 if the targeted CPU has 64-bit address registers; if the targeted CPU does not have 64–bit address registers, the macro is not defined.
`_LD64`	Defined as 1 if the `long double` basic type has 64 bits of precision; if 128-bit precision is used, the macro is not defined.
`_FASTMD`	Defined as 1 if the fast multiply/divide sequence is enabled; if the machine type is Cray T3E or if fast multiply/divide is not used, the macro is not defined.
`_MAXVL`	Defined as the maximum hardware vector length (64 or 128); if the machine type is Cray T3E, the macro is not defined.

6.4 Macros Based on the Compiler

The following macros provide information about compiler features:

Macro	Description
_RELEASE	Defined as the major release level of the compiler.
_CRAYC	Defined as 1 to identify the Cray Standard C compiler.

Debugging C/C++ Code [7]

The Cray TotalView symbolic debugger is available to help you debug C and C++ codes. In addition, the Cray C and C++ compilers provide the following features to help you in debugging codes:

- The `-G` and `-g` compiler options provide symbol information about your source code for use by the Cray TotalView debugger. For more information on these compiler options, see Section 2.17.1, page 26.
- The `-h [no]trunc` option helps identify numerically unstable algorithms. For more information, see Section 2.15.7, page 25.
- The `-h [no]bounds` option and the `#pragma _CRI [no]bounds` directive let you check pointer and array references. The `-h [no]bounds` option is described in Section 2.17.2, page 27. The `#pragma _CRI [no]bounds` directive is described in Section 4.5.2, page 64.
- The `#pragma _CRI message` directive lets you add warning messages to sections of code where you suspect problems. The `#pragma _CRI message` directive is described in Section 4.5.4, page 66.
- The `#pragma _CRI [no]opt` directive lets you selectively isolate portions of your code to optimize, or to toggle optimization on and off in selected portions of your code. The `#pragma _CRI [no]opt` directive is described in Section 4.5.5, page 66.

7.1 Cray TotalView Debugger

The Cray TotalView debugger is designed for use with C, C++, or Fortran source code and is available on all Cray Research systems. The TotalView debugger is documented in *Introducing the Cray TotalView Debugger*.

Some of the functions available in the Cray TotalView debugger allow you to perform the following actions:

- Set and clear breakpoints, which can be conditional, at both the source code level and the assembly code level
- Examine core files
- Step through a program, including across function calls

- Reattach to the executable file after editing and recompiling
- Edit values of variables and memory locations
- Evaluate code fragments

7.2 Compiler Debugging Options

To use the Cray TotalView debugger in debugging your code, you must first compile your code using one of the debugging options (`-g` or `-G`). These options are specified as follows:

- `-Gf`

  If you specify the `-Gf` debugging option, the Cray TotalView debugger allows you to set breakpoints at function entry and exit and at labels.

- `-Gp`

  If you specify the `-Gp` debugging option, the Cray TotalView debugger allows you to set breakpoints at function entry and exit, labels, and at places where execution control flow changes (for example, loops, `switch`, and `if...else` statements).

- `-Gn` or `-g`

  If you specify the `-Gn` or `-g` debugging option, the Cray TotalView debugger allows you to set breakpoints at function entry and exit, labels, and executable statements. These options force all compiler optimizations to be disabled as if you had specified `-O0`.

Users of the Cray C and C++ compilers do not have to sacrifice run-time performance to debug codes. Many compiler optimizations are inhibited by breakpoints generated for debugging. By specifying a higher debugging level, fewer breakpoints are generated and better optimization occurs.

However, consider the following cases in which optimization is affected by the `-Gp` and `-Gf` debugging options:

- Vectorization can be inhibited if a label exists within the vectorizable loop.
- Vectorization can be inhibited if the loop contains a nested block and the `-Gp` option is specified.
- When the `-Gp` option is specified, setting a breakpoint at the first statement in a vectorized loop allows you to stop and display at each vector iteration.

However, setting a breakpoint at the first statement in an unrolled loop may not allow you to stop at each vector iteration.

Interlanguage Communication [8]

In some situations, it is necessary or advantageous to make calls to assembly or Fortran functions from C or C++ programs. This section describes how to make such calls. It also discusses calls to C and C++ functions from Fortran and assembly language. For additional information on interlanguage communication, see *Interlanguage Programming Conventions*. The calling sequence is described in detail on the `callseq`(3) man page, which is included in the *Application Programmer's Library Reference Manual*.

8.1 Interlanguage Communication with Cray Standard C and Cray C++

Cray Standard C and Cray C++ provide a mechanism for declaring external functions that are written in other languages. This allows the C/C++ programmer to write portions of an application in C, C++, Fortran, or assembly language. This can be useful in cases where these other languages provide performance advantages or utilities that are not available in C or C++.

This section describes how to call assembly language and Fortran programs from a C or C++ program. It also discusses the issues related to calling C or C++ programs from other languages. These calls apply to UNICOS and UNICOS/mk systems unless stated otherwise.

8.1.1 Calling Assembly Language Functions from a C or C++ Function

You can sometimes avoid bottlenecks in programs by rewriting parts of the program in assembly language, maximizing performance by selecting instructions to reduce machine cycles. When writing assembly language functions that will be called by C or C++ functions, use the standard UNICOS program linkage macros. When using these macros, you do not need to know the specific registers used by the C or C++ program or by the calling sequence of the assembly coded routine. UNICOS program linkage macros are described in the *UNICOS Macros and Opdefs Reference Manual*.

In C++, use `extern "C"` to declare the assembly language function. In C++, the `main` function must be written in C++.

8.1.1.1 Cray Assembly Language (CAL) Functions on UNICOS Systems

The use of Cray Assembly Language (CAL) on UNICOS systems is described in the *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*.

On UNICOS systems, the `ALLOC`, `DEFARG`, `DEFB`, `DEFT`, `ENTER`, `EXIT`, `MXCALLEN`, and `PROGRAM` macros can be used to define the calling list; B and T register use; temporary storage; and entry and exit points.

8.1.1.2 Cray Assembler for MPP (CAM) Functions on UNICOS/mk Systems

The use of the Cray Assembler for MPP (CAM) on UNICOS/mk systems is described in the *Cray Assembler for MPP (CAM) Reference Manual*.

On UNICOS/mk systems, the `ALLOC`, `LOAD`, `STORE` and `DEFARG`, `ENTER`, `EXIT`, `ADDRESS`, `VALUE` macros can be used to define local (temporary) storage; entry or exit points; argument processing; and calls to other functions.

8.1.2 Calling Fortran Functions and Subroutines from a C or C++ Function

This subsection describes the following aspects of calling Fortran from C or C++: requirements and guidelines, MPP considerations, argument passing, array storage, logical and character data, and accessing blank common from C and C++ programs.

8.1.2.1 Requirements

Keep the following points in mind when calling Fortran functions from C/C++:

- Fortran uses the call-by-address convention, and C/C++ uses the call-by-value convention, which means that only pointers should be passed to Fortran subprograms. See Section 8.1.2.2, page 109.
- Fortran arrays are in column-major order, and C/C++ arrays are in row-major order. This indicates which dimension is indicated by the first value in an array element subscript. See Section 8.1.2.3, page 109.
- Single-dimension arrays of integers and single-precision floating-point numbers are the only aggregates that can be passed as parameters without changing the arrays.
- Fortran character pointers and C/C++ character pointers are incompatible. See Section 8.1.2.4, page 111.

- Fortran logical values and C/C++ Boolean values are not fully compatible. See Section 8.1.2.4, page 111.
- External C/C++ variables are stored in common blocks of the same name, making them readily accessible from Fortran programs if the C/C++ variable is uppercase.
- When declaring Fortran functions or objects in C/C++, the name must be specified in all uppercase letters, digits, or underscore characters and consist of 31 or fewer characters.
- In C, Fortran functions can be declared using the `fortran` keyword (see Section 3.6, page 57). The `fortran` keyword is not available in C++. Instead, fortran functions must be declared by specifying `extern "C"`.
- In C++, the `main` function must be written in C++.
- On UNICOS/mk systems, the C/C++ language `float` type does not match the Fortran `REAL` type. The `float` type is 32 bits for C/C++ on UNICOS/mk systems; the Fortran `REAL` type is 64 bits. However, Fortran includes the `REAL*4` type that matches the C language `float` type.

8.1.2.2 Argument Passing

Because Fortran subroutines expect arguments to be passed by pointers rather than by value, C and C++ functions called from Fortran subroutines must pass pointers rather than values.

All argument passing in C is strictly by value. To prepare for a function call between two C functions, a copy is made of each actual argument. A function can change the values of its formal parameters, but these changes cannot affect the values of the actual arguments. It is possible, however, to pass a pointer. (All array arguments are passed by this method.) This capability is analogous to the Fortran method of passing arguments.

In addition to passing by value, C++ also provides passing by reference.

8.1.2.3 Array Storage

C and C++ arrays are stored in memory in row-major order; and Fortran arrays are stored in memory in column-major order. For example, the C/C++ array declaration `int A[3][2]` is stored in memory as:

A[0][0]	A[0][1]
A[1][0]	A[1][1]
A[2][0]	A[2][1]

The previously defined array is viewed linearly in memory as:

```
A[0][0] A[0][1] A[1][0] A[1][1] A[2][0] A[2][1]
```

The Fortran array declaration `INTEGER A(3,2)` is stored in memory as:

A(1,1)	A(2,1)	A(3,1)
A(1,2)	A(2,2)	A(3,2)

The previously defined array is viewed linearly in memory as:

```
A(1,1)  A(2,1)  A(3,1)  A(1,2)  A(2,2)  A(3,2)
```

When an array is shared between C/C++ and Fortran, its dimensions are declared and referenced in C/C++ in the opposite order in which they are declared and referenced in Fortran. Since arrays are zero-based in C/C++ and one-based in Fortran, in C/C++ you should subtract 1 from the array subscripts that you would normally use in Fortran.

For example, using the Fortran declaration of array A in the preceding example, the equivalent declaration in C/C++ would be:

```
int a[2][3];
```

The following table shows how to access elements of the array from Fortran and C/C++:

Fortran	C/C++
A(1,1)	A[0][0]
A(2,1)	A[0][1]
A(3,1)	A[0][2]
A(1,1)	A[1][0]
A(2,2)	A[1][1]
A(3,2)	A[1][2]

8.1.2.4 Logical and Character Data

Logical and character data need special treatment for calls between C/C++ and Fortran. Fortran has a character descriptor that is incompatible with a character pointer in C/C++. The techniques used to represent logical (Boolean) values also differ between C/C++ and Fortran.

Mechanisms you can use to convert one type to the other are provided by the standard header file and conversion utilities shown in the following list.

Header file or utility	Description
`<fortran.h>`	Header file that defines the type `_fcd`, which maps to the Fortran character descriptor and defines or declares the macros or functions contained in this list.
`_cptofcd`	Conversion utility that converts a C/C++ character pointer to a Fortran character descriptor.
`_fcdtocp`	Conversion utility that converts a Fortran character descriptor to a C/C++ character pointer.
`_fcdlen`	Conversion utility that extracts the byte length from the Fortran character descriptor. Because Fortran does not terminate character

	strings with a null character, `_fcdlen` can be used to determine the last character in the string.
`_btol`	Conversion utility that converts a 0 to a Fortran logical `.FALSE.` and a nonzero value to a Fortran logical `.TRUE.`
`_ltob`	Conversion utility that converts a Fortran logical `.FALSE.` to a 0 and a Fortran logical `.TRUE.` to a 1.

For more information on these utilities, see the description of the `_cptofcd`(3) function in the *UNICOS System Libraries Reference Manual*.

8.1.2.5 Accessing Named Common from C/C++

The following example demonstrates how external C variables are accessible in Fortran named common blocks. It shows a C function calling a Fortran subprogram, the associated Fortran subprogram, and the associated input and output.

In this example, the C structure `ST` is accessed in the Fortran subprogram as common block `ST`. The name of the C structure and the Fortran common block must match. Note that this requires that the C structure name be uppercase. The C structure member names and the Fortran common block member names do not have to match, as is shown in this example.

The following C main program calls the Fortran subprogram `FCTN`:

```
#include <stdio.h>
struct
{
   int i;
   double a[10];
   long double d;
} ST;

main()
{
   int i;

   /* initialize struct ST */
   ST.i = 12345;

   for (i = 0; i < 10; i++)
      ST.a[i] = i;

   ST.d = 1234567890.1234567890L;

   /* print out the members of struct ST */
   printf("In C: ST.i = %d, ST.d = %20.10Lf\n", ST.i, ST.d);
   printf("In C: ST.a = ");
   for (i = 0; i < 10; i++)
      printf("%4.1f", ST.a[i]);
   printf("\n\n");

   /* call the fortran function */
   FCTN();
}
```

The following example is the Fortran subprogram `FCTN` called by the previous C main program:

```
C *********** Fortran subprogram (f.f): ***********

      SUBROUTINE FCTN

      COMMON /ST/STI, STA(10), STD
      INTEGER STI
      REAL STA
      DOUBLE PRECISION STD
```

```
      INTEGER I

      WRITE(6,100) STI, STD
  100 FORMAT ('IN FORTRAN: STI = ', I5, ', STD = ', D25.20)
      WRITE(6,200) (STA(I), I = 1,10)
  200 FORMAT ('IN FORTRAN: STA =', 10F4.1)
      END
```

The previous C and Fortran examples are executed by the following commands and produce the output shown:

```
$ cc -c c.c
$ f90 -c f.f
$ segldr c.o f.o
$ a.out
 ST.i = 12345, ST.d = 1234567890.1234567890
 In C: ST.a =  0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0

 IN FORTRAN: STI = 12345, STD = .12345678901234567889D+10
 IN FORTRAN: STA = 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
$
```

8.1.2.6 Accessing Blank Common from C/C++

Fortran includes the concept of a common block. A *common block* is an area of memory that can be referenced by any program unit in a program. A *named common block* has a name specified in a Fortran COMMON or TASKCOMMON statement, along with the names of variables or arrays stored in the block. A *blank common block*, sometimes referred to as blank common, is declared in the same way, but without a name.

There is no way to access blank common from C/C++ similar to accessing a named common block. However, you can write a simple Fortran function to return the address of the first word in blank common to the C/C++ program and then use that as a pointer value to access blank common.

The following example shows how Fortran blank common can be accessed using C/C++ source code:

```
#include <stdio.h>

struct st
{
  float a;
  float b[10];
} *ST;

#ifdef __cplusplus
  extern "C" struct st *MYCOMMON(void);
  extern "C" void FCTN(void);
#else
  fortran struct st *MYCOMMON(void);
  fortran void FCTN(void);
#endif

main()
{
    int i;

    ST = MYCOMMON();
    ST->a = 1.0;
    for (i = 0; i < 10; i++)
        ST->b[i] = i+2;
    printf("\n In C/C++\n");
    printf("     a = %5.1f\n", ST->a);
    printf("     b = ");
    for (i = 0; i < 10; i++)
        printf("%5.1f ", ST->b[i]);
    printf("\n\n");

    FCTN();
}
```

The following Fortran source code accesses blank common and is accessed from the C/C++ source code in the previous example:

```
SUBROUTINE FCTN
COMMON // STA,STB(10)
PRINT *, "IN FORTRAN"
PRINT *, "    STA = ",STA
PRINT *, "    STB = ",STB
STOP
END

FUNCTION MYCOMMON()
COMMON // A
MYCOMMON = LOC(A)
RETURN
END
```

The output of the previous C/C++ source code is as follows:

```
In C
  a =  1.0
  b = 2.0   3.0   4.0   5.0   6.0   7.0   8.0   9.0  10.0  11.0
```

The output of the previous Fortran source code is as follows:

```
IN FORTRAN
  STA = 1.
  STB = 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.
```

8.1.2.7 C and Fortran Example

The following example illustrates a C function that calls a Fortran subprogram. The Fortran subprogram follows the C function and the input and output from this sequence follows the Fortran subprogram.

```
/*                 C program (main.c):                   */

#include <stdio.h>
#include <string.h>
#include <fortran.h>

fortran double FTNFCTN (_fcd, int *);

double FLOAT1 = 1.6;
double FLOAT2;  /* Initialized in FTNFCTN */

main()
{
     int clogical, ftnlogical, cstringlen;
     double rtnval;
     char *cstring = "C Character String";
      _fcd ftnstring;

/* Convert cstring and clogical to their Fortran equivalents */
     ftnstring = _cptofcd(cstring, strlen(cstring));
     clogical = 1;
     ftnlogical = _btol(clogical);

/* Print values of variables before call to Fortran function */
     printf(" In main: FLOAT1 = %g; FLOAT2 = %g\n",
          FLOAT1, FLOAT2);
     printf(" Calling FTNFCTN with arguments:\n");
     printf(" string = \"%s\"; logical = %d\n\n", cstring, clogical);

     rtnval = FTNFCTN(ftnstring, &ftnlogical);

/* Convert ftnstring and ftnlogical to their C equivalents */
     cstring = _fcdtocp(ftnstring);
     cstringlen = _fcdlen(ftnstring);
     clogical = _ltob(&ftnlogical);

/* Print values of variables after call to Fortran function */
     printf(" Back in main: FTNFCTN returned %g\n", rtnval);
     printf(" and changed the two arguments:\n");
     printf(" string = \"%.*s\"; logical = %d\n",
     cstringlen, cstring, clogical);
}
```

```
C                  Fortran subprogram (ftnfctn.f):


     FUNCTION FTNFCTN(STR, LOG)

     REAL FTNFCTN
     CHARACTER*(*) STR
     LOGICAL LOG

     COMMON /FLOAT1/FLOAT1
     COMMON /FLOAT2/FLOAT2
     REAL FLOAT1, FLOAT2
     DATA FLOAT2/2.4/          ! FLOAT1 INITIALIZED IN MAIN

C      PRINT CURRENT STATE OF VARIABLES
       PRINT*, '     IN FTNFCTN: FLOAT1 = ', FLOAT1,
      1                          ';FLOAT2 = ', FLOAT2
       PRINT*, '     ARGUMENTS:    STR = "', STR, '"; LOG = ', LOG

C      CHANGE THE VALUES FOR STR(ING) AND LOG(ICAL)
       STR = 'New Fortran String'
       LOG = .FALSE.

       FTNFCTN = 123.4
       PRINT*, '    RETURNING FROM FTNFCTN WITH ', FTNFCTN
       PRINT*
       RETURN
       END
```

The previous C function and Fortran subprogram are executed by the following commands and produce the following output:

```
$ cc -c main.c
$ f90 -c ftnfctn.f
$ segldr main.o ftnfctn.o
$ a.out
$
In main: FLOAT1 = 1.6;  FLOAT2 = 2.4
Calling FTNFCTN with arguments:
string = "C Character String"; logical = 1

IN FTNFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
ARGUMENTS:   STR = "C Character String"; LOG = T
RETURNING FROM FTNFCTN WITH 123.4

Back in main: FTNFCTN returned 123.4
and changed the two arguments:
string = "New Fortran String"; logical = 0
$
```

8.1.2.8 Calling a Fortran Program from a C++ Program

The following example illustrates how a Fortran program can be called from a C++ program:

```
#include <iostream.h>
extern "C" int FORTRAN_ADD_INTS(int *arg1, int &arg2);

main()
{
    int num1, num2, res;
    cout << "Start C++ main" << endl << endl;

    //Call FORTRAN function to add two integers and return result.
    //Note that the second argument is a reference parameter so
    //it is not necessary to take the address of the
    //variable num2.

    num1 = 10;
    num2 = 20;
    cout << "Before Call to FORTRAN_ADD_INTS" << endl;
    res = FORTRAN_ADD_INTS(&num1, num2);
    cout << "Result of FORTRAN Add = " << res << endl << endl;
    cout << "End C++ main" << endl;
}
```

The Fortran program that is called from the C++ main function in the preceding example is as follows:

```
INTEGER FUNCTION FORTRAN_ADD_INTS(Arg1, Arg2)
INTEGER Arg1, Arg2

PRINT *," FORTRAN_ADD_INTS, Arg1,Arg2 = ", Arg1, Arg2
FORTRAN_ADD_INTS = Arg1 + Arg2
END
```

The output from the execution of the preceding example is as follows:

```
Start C++ main

Before Call to FORTRAN_ADD_INTS
  FORTRAN_ADD_INTS, Arg1,Arg2 =  10,  20
Result of FORTRAN Add = 30

End C++ main
```

8.1.3 Calling a C/C++ Function from an Assembly Language or Fortran Program

A C/C++ function can be called from Fortran or assembly language. When calling from Fortran, keep in mind the information in Section 8.1.2, page 108.

When calling a C++ function from Fortran or assembly language, the C++ function must be declared with extern "C" storage class, the main function must be written in C++, and the program must be linked with the CC command. C++ main is responsible for initializing the static constructors for C++ functions.

The example that follows illustrates a Fortran program that calls a C function. The C function being called, the commands required, and the associated input and output are also included.

```
C   Fortran program (main.f):

      PROGRAM MAIN

      REAL CFCTN
      COMMON /FLOAT1/FLOAT1
      COMMON /FLOAT2/FLOAT2
      REAL FLOAT1, FLOAT2
      DATA FLOAT1/1.6/      ! FLOAT2 INITIALIZED IN cfctn
      LOGICAL LOG
      CHARACTER*24 STR
      REAL RTNVAL

C  INITIALIZE VARIABLES STR(ING) AND LOG(ICAL)
      STR = 'Fortran Character String'
      LOG = .TRUE.

C  PRINT VALUES OF VARIABLES BEFORE CALL TO C FUNCTION
      PRINT*, ' IN MAIN: FLOAT1 = ', FLOAT1,
     1                 '; FLOAT2 = ', FLOAT2
      PRINT*, ' CALLING CFCTN WITH ARGUMENTS: '
      PRINT*, ' STR = "', STR, '"; LOG = ', LOG
      PRINT*

      RTNVAL = CFCTN(STR, LOG)

C  PRINT VALUES OF VARIABLES AFTER CALL TO C FUNCTION
      PRINT*, ' BACK IN MAIN: CFCTN RETURNED ', RTNVAL
      PRINT*, ' AND CHANGED THE TWO ARGUMENTS: '
      PRINT*, ' STR = "', STR, '"; LOG = ', LOG
      END
```

The following example illustrates the associated C function that is being called:

```
/*               C function (cfctn.c):                      */
#include <fortran.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

double FLOAT1;        /* Initialized in MAIN */
double FLOAT2 = 2.4;

double CFCTN(_fcd str, int *log)
{
     int slen;
     int clog;
     float returnval;
     char *cstring;
     char newstr[25];

/* Convert str and log passed from Fortran MAIN */
/* into C equivalents */
    slen = _fcdlen(str);
    cstring = malloc(slen+1);
    strncpy(cstring, _fcdtocp(str), slen);
    cstring[slen] = '\0';
    clog = _ltob(log);

/* Print the current state of the variables */
    printf("     In CFCTN: FLOAT1 = %.1f; FLOAT2 = %.1f\n",
               FLOAT1, FLOAT2);
    printf("     Arguments: str = "%s"; log = %d\n",
    cstring, clog);

/* Change the values for str and log */
    strncpy(_fcdtocp(str), "C Character String     ", 24);
    *log = 0;

    returnval = 123.4;
    printf("     Returning from CFCTN with %.1f\n\n", returnval);
    return(returnval);
}
```

The previous Fortran program and C function are executed by the following commands and produce the following output:

```
$ cc -c cfctn.c
$ f90 -c main.f
$ f90 cfctn.o main.o
$ a.out
$
IN MAIN: FLOAT1 = 1.6; FLOAT2 = 2.4
CALLING CFCTN WITH ARGUMENTS:
STR = "Fortran Character String"; LOG = T

     In CFCTN: FLOAT1 = 1.6; FLOAT2 = 2.4
     Arguments: str = "Fortran Character String"; log = 1
     Returning from CFCTN with 123.4

BACK IN MAIN: CFCTN RETURNED 123.4
AND CHANGED THE TWO ARGUMENTS:
STR = "C Character String  "; LOG = F
$
```

8.1.4 Calls between C and C++ Functions

The following requirements must be considered when making calls between functions written in C and C++:

- In C++, the `extern "C"` storage class is required when declaring an external function that is written in C or when declaring a C++ function that is to be called from C. Normally the C++ compiler will mangle function names to encode information about the function's prototype in the external name. This prevents direct access to these function names from a C function. The `extern "C"` keyword will prevent the C++ compiler from performing name mangling.
- The `main` function must be a C++ program.
- The program must be linked using the `CC` command.

Objects can be shared between C and C++. There are some C++ objects that are not accessible to C functions (such as classes). The following object types can be shared directly:

- Integral and floating types.
- Structures and unions that are declared identically in C and C++. In order for structures and unions to be shared, they must be declared with identical members in the identical order.

- Arrays and pointers to the above types.

In the following example, a C function (`C_add_func`) is called by the C++ `main` function:

```
        C++ Main Program

#include <iostream.h>

extern "C" int C_add_func(int, int);
int global_int = 123;

main()
{
  int res, i;

  cout << "Start C++ main" << endl;

   // Call C function to add two integers and return result.

  cout << "Call C C_add_func" << endl;
  res = C_add_func(10, 20);
  cout << "Result of C_add_func = " << res << endl;
  cout << "End C++ main << endl;
}
```

The C function (`C_add_func`) is as follows:

```
#include <stdio.h>

extern int global_int;

int C_add_func(int p1, int p2)
{
    printf("\tStart C function C_add_func.\n");
    printf("\t\tp1     = %d\n", p1);
    printf("\t\tp2     = %d\n", p2);
    printf("\t\tglobal_int = %d\n", global_int);
    return p1 + p2;
}
```

The output from the execution of the calling sequence illustrated in the preceding example is as follows:

```
Start C++ main
Call C C_add_func
        Start C function C_add_func.
                p1      = 10
                p2      = 20
                global_int = 123
Result of C_add_func = 30
End C++ main
```

Implementation-defined Behavior [9]

This section describes compiler behavior that is defined by the implementation according to the C and/or C++ standards. The standards require that the behavior of each particular implementation be documented.

9.1 Implementation-defined Behavior

The C and C++ standards define implementation-defined behavior as behavior, for a correct program construct and correct data, that depends on the characteristics of the implementation. The behavior of the Cray Standard C and Cray C++ compilers for these cases is summarized in this section.

9.1.1 Messages

All diagnostic messages issued by the Cray compilers are reported through the UNICOS message system. For information on messages issued by the compilers and for information about the UNICOS message system, see Appendix C, page 157.

9.1.2 Environment

When `argc` and `argv` are used as parameters to the `main` function, the array members `argv[0]` through `argv[argc-1]` contain pointers to strings that are set by the command shell. The shell sets these arguments to the list of words on the command line used to invoke the compiler (the argument list). For further information on how the words in the argument list are formed, refer to the documentation on the shell in which you are running. For information on UNICOS shells, see the `sh`(1), `csh`(1), or `ksh`(1) man pages.

A third parameter, `char **envp`, provides access to environment variables. The value of the parameter is a pointer to the first element of an array of null-terminated strings, that matches the output of the `env`(1) command. The array of pointers is terminated by a null pointer.

The compiler does not distinguish between interactive devices and other, noninteractive devices. The library, however, may determine that `stdin`, `stdout`, and `stderr` (`cin`, `cout`, and `cerr` in C++) refer to interactive devices and buffer them accordingly. For further information, see the description of I/O in the *UNICOS System Libraries Reference Manual*.

9.1.2.1 Identifiers

The *identifier* (as defined by the standards) is merely a sequence of letters and digits. Specific uses of identifiers are called *names*.

In C, the compiler treats the first 255 characters of a name as significant, regardless of whether it is an internal or external name. The case of names, including external names, is significant. In C++, all characters of a name are significant.

9.1.2.2 Types

Table 3 summarizes data types supported on Cray Research systems and the characteristics of each type. *Representation* is the number of bits used to represent values for each data type. *Memory* is the number of storage bits that the data type occupies.

For the Cray Research implementation, *size*, in the context of the `sizeof` operator, refers to the size allocated to store the operands in memory; it does not refer to representation, as specified in Table 3. Thus, the `sizeof` operator will return a size that is equal to the value in the Memory column of Table 3 divided by 8 (the number of bits in a byte).

Table 3. Cray Research systems data type mapping

	Cray PVP systems		Cray MPP systems	
Type	Representation (bits)	Memory (bits)	Representation (bits)	Memory (bits)
bool (C++ only)	8	8	8	8
char	8	8	8	8
wchar_t (C++ only)	64	64	64	64
short	32 Cray T90: 46/64 (See Footnote 1)	64	32	32
int	46/64 (See Footnote 1)	64	64	64
long	64	64	64	64
long long (See Footnote 2)	64	64	64	64
float	64	64	32	32
double	64	64	64	64
long double	128	128	64	64
float complex (See Footnote 3)	128 (64 each part)	128	64 (32 each part)	64
double complex (See Footnote 3)	128 (64 each part)	128	128 (64 each part)	128
long double complex (See Footnote 3)	256 (128 each part)	256	128 (64 each part)	128
void and char pointers	64	64	64	64

	Cray PVP systems		Cray MPP systems	
Type	Representation (bits)	Memory (bits)	Representation (bits)	Memory (bits)
Other pointers	32 Cray T90: 64	64	64	64

Footnote 1: Depends on use of the `-h [no]fastmd` option. This option is described in Section 2.15.2, page 23.
Footnote 2: Available in extended mode only.
Footnote 3: Cray Research extension (Cray Standard C only).

9.1.2.3 Characters

The full 8-bit ASCII code set can be used in source files. Characters not in the character set defined in the standard are permitted only within character constants, string literals, and comments. The `-h [no]calchars` option allows the use of the `@` sign and `$` sign in identifier names. For more information on the `-h [no]calchars` option, see Section 2.7.3, page 11.

A character consists of 8 bits. Up to 8 characters can be packed into a Cray word. A plain `char` type, one that is declared without a `signed` or `unsigned` keyword, is treated as an unsigned type.

Character constants and string literals can contain any characters defined in the 8-bit ASCII code set. The characters are represented in their full 8-bit form. A character constant can contain up to 8 characters. The integer value of a character constant is the value of the characters packed into a word from left to right, with the result right-justified, as shown in the following table:

Character constant	Integer value
`'a'`	`0x61`
`'ab'`	`0x6162`

In a character constant or string literal, if an escape sequence is not recognized, the \ character that initiates the escape sequence is ignored, as shown in the following table:

Character constant	Integer value	Explanation
'\a'	0x7	Recognized as the ASCII BEL character
'\8'	0x38	Not recognized; ASCII value for 8
'\['	0x5b	Not recognized; ASCII value for [
'\c'	0x63	Not recognized; ASCII value for c

9.1.2.4 Wide Characters

Wide characters are treated as signed 64-bit integer types. Wide character constants cannot contain more than one multibyte character. Multibyte characters in wide character constants and wide string literals are converted to wide characters in the compiler by calling the `mbtowc`(3) function. The current locale in effect at the time of compilation determines the method by which `mbtowc` converts multibyte characters to wide characters, and the shift states required for the encoding of multibyte characters in the source code. If a wide character, as converted from a multibyte character or as specified by an escape sequence, cannot be represented in the extended execution character set, it is truncated.

9.1.2.5 Integers

All integral values are represented in a twos complement format. For representation and memory storage requirements for integral types on Cray Research systems, see Table 3, page 129.

When an integer is converted to a shorter signed integer, and the value cannot be represented, the result is the truncated representation treated as a signed quantity. When an unsigned integer is converted to a signed integer of equal length, and the value cannot be represented, the result is the original representation treated as a signed quantity.

The bitwise operators (unary operator ~ and binary operators `<<`, `>>`, `&`, `^`, and `|`) operate on signed integers in the same manner in which they operate on unsigned integers. The result of `E1 >> E2`, where `E1` is a negative-valued signed integral value, is `E1` right-shifted `E2` bit positions; vacated bits are filled with 0's on UNICOS systems and 1's on UNICOS/mk systems. On UNICOS/mk systems, this behavior can be modified by using the `-h nosignedshifts` option (see Section 2.7.4, page 11).

On UNICOS/mk systems, the shift operators (>> and <<) use only the rightmost six bits of the second operand. For example, shifting by 65 is the same as shifting by 1. On Cray Y-MP systems, bits higher than the sixth bit are not ignored. Values higher than 63 cause the result to be 0.

The result of the / operator is the largest integer less than or equal to the algebraic quotient when either operand is negative and the result is a nonnegative value. If the result is a negative value, it is the smallest integer greater than or equal to the algebraic quotient. The / operator behaves the same way in C/C++ as in Fortran.

The sign of the result of the % operator is the sign of the first operand.

Integer overflow is ignored. Because some integer arithmetic uses the floating-point instructions on UNICOS systems, floating-point overflow can occur during integer operations. Division by 0 and all floating-point exceptions, if not detected as an error by the compiler, can cause a run-time abort.

9.1.2.6 Floating-point Arithmetic

Cray Research systems use either Cray floating-point arithmetic or IEEE floating-point arithmetic. These types of floating-point representation are described in the sections that follow.

9.1.2.6.1 Cray Floating-point Representation

Types `float` and `double` represent Cray single-precision (64-bit) floating-point values; `long double` represents Cray double-precision (128-bit) floating-point values.

An integral number that is converted to a floating-point number that cannot exactly represent the original value is truncated toward 0. A floating-point number that is converted to a narrower floating-point number is also truncated toward 0.

Floating-point arithmetic depends on implementation-defined ranges for types of data. The values of the minimums and maximums for these ranges are defined by macros in the standard header file `float.h`. All floating-point operations on operands that are within the defined range yield results that are also in this range if the true mathematical result is in the range. The results are accurate to within the ability of the hardware to represent the true value.

The maximum positive value for types `float`, `double`, and `long double` is approximately as follows:

2.7×10^{2456}

Several math functions return this upper limit if the true value equals or exceeds it.

The minimum positive value for types `float`, `double`, and `long double` is approximately as follows:

3.67×10^{-2466}

These numbers define a range that is slightly smaller than the value that can be represented by Cray Research hardware, but use of numbers outside this range may not yield predictable results. For exact values, use the values defined in the header file, `float.h`.

A floating-point value, when rounded off, can be accurately represented to approximately 14 decimal places for types `float` and `double`, and to approximately 28 decimal places for type `long double` as determined by the following equation:

$$number\ of\ decimal\ digits = \frac{number\ of\ bits}{\log_2 10.0} \tag{9.1}$$

Digits beyond these precisions may not be accurate. It is safest to assume only 14 or 28 decimal places of accuracy.

Epsilon, the difference between 1.0 and the smallest value greater than 1.0 that is representable in the given floating-point type, is approximately 7.1×10^{-15} for types `float` and `double`, and approximately 2.5×10^{-29} for type `long double`.

9.1.2.6.2 IEEE Floating-point Representation

On UNICOS/mk systems, `float` represents IEEE single-precision (32-bit) floating-point values; `double` and `long double` represent double-precision (64-bit) floating-point values. IEEE extended double precision (128–bit) is not available on UNICOS/mk systems.

On UNICOS systems with IEEE floating-point hardware, `float` and `double` represent IEEE double-precision (64-bit) floating-point values. The `long double` represents IEEE extended double-precision (128-bit) floating-point values. IEEE single-precision (32-bit) is not available on UNICOS systems.

An integral number that is converted to a floating-point number that cannot exactly represent the original value is rounded according to the current rounding mode. A floating-point number that is converted to a floating-point number with fewer significant digits also is rounded according to the current rounding mode on UNICOS/mk systems; on UNICOS systems, the number is rounded to closest, but not in an IEEE round-to-nearest fashion.

Floating-point arithmetic depends on implementation-defined ranges for types of data. The values of the minimums and maximums for these ranges are defined by macros in the standard header file, `float.h`. All floating-point operations on operands that are within the defined range yield results that are also in this range if the true mathematical result is in the range. The results are accurate to within the ability of the hardware to represent the true value.

The maximum positive values are approximately as follows:

3.4×10^{38}	Single (32 bits)
1.8×10^{308}	Double (64 bits)
1.2×10^{4932}	Extended double (128 bits)

The minimum positive values are approximately as follows:

1.8×10^{-38}	Single (32 bits)
2.2×10^{-308}	Double (64 bits)
3.4×10^{-4932}	Extended double (128 bits)

For exact values, use the macros defined in the header file, `float.h`.

Rounding of 32 and 64 bit floating-point arithmetic is determined by the current rounding mode. The 128 bit floating-point arithmetic is rounded to the closest, without regard to the rounding mode. A floating-point value, when rounded off, can be accurately represented to approximately 7 decimal places for single-precision types, approximately 16 decimal places for double-precision types, and approximately 34 decimal places for extended double-precision types as determined by the following equation:

$$number\ of\ decimal\ digits = \frac{number\ of\ bits}{\log_2 10.0} \tag{9.2}$$

Digits beyond these precisions may not be accurate.

Epsilon, the difference between 1.0 and the smallest value greater than 1.0 that is representable in the given floating-point type, is approximately as follows:

1.2×10^{-7}	Single (32 bits)
2.2×10^{-16}	Double (64 bits)
1.9×10^{-34}	Extended double (128 bits)

Upon entering the `main` function at the beginning of the program execution, the rounding mode is set to round to nearest, all floating-point exception status flags are cleared, and traps are enabled for overflow, invalid operation, and division-by-zero exceptions. Traps are disabled for all other exceptions. On Cray T90 systems with IEEE floating-point hardware the default rounding mode and the trap modes can be specified at program startup by using the `cpu`(8) command (see the `cpu` man page for more information).

9.1.2.7 Arrays and Pointers

An `unsigned int` value can hold the maximum size of an array. The type `size_t` is defined to be a `typedef` name for `unsigned int` in the headers: `malloc.h`, `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, and `time.h`. If more than one of these headers is included, only the first defines `size_t`.

A type `int` can hold the difference between two pointers to elements of the same array. The type `ptrdiff_t` is defined to be a `typedef` name for `int` in the header `stddef.h`.

On all Cray Research systems, if a pointer type's value is cast to a signed or unsigned `int` or `long int`, and then cast back to the original type's value, the two pointer values will compare equal.

Pointers on UNICOS systems differ from pointers on UNICOS/mk systems. The sections that follow describe pointer implementation on each type of system.

9.1.2.7.1 Pointers on UNICOS Systems

Although a pointer value can be stored in an object of integer type, an operation may give different results when performed on the same value treated as an integer or as a pointer. An integer result should not be used as a pointer. For example, do not assume that adding 5 to an integer is the same as adding 5 to a pointer, because the result is affected by the kind of pointer used in the operation. In particular, results may differ from those on a system using a simpler representation of pointers, such as UNICOS/mk systems.

Pointers other than character pointers are internally represented just like integers: as a single 64-bit field. Character pointers use one of the formats shown in Figure 1, depending on the size of A registers.

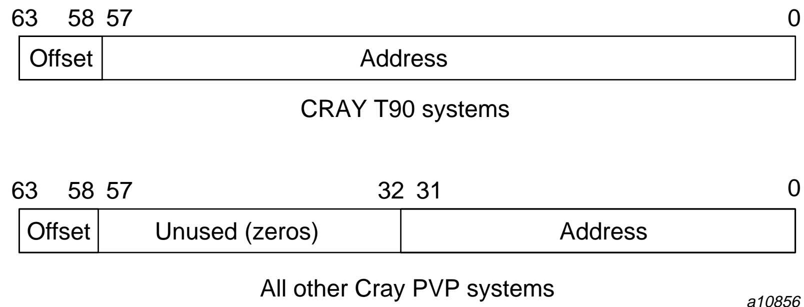

Figure 1. Character pointer format

Converting a 64-bit integer to a character pointer type results in a pointer to the byte specified by the value in the offset field of the word specified in the address field.

9.1.2.7.2 Pointers on UNICOS/mk systems

Pointers on UNICOS/mk systems are byte pointers. Byte pointers use the same internal representation as integers; a byte pointer counts the numbers of bytes from the first address.

A pointer can be explicitly converted to any integral type large enough to hold it. The result will have the same bit pattern as the original pointer. Similarly, any value of integral type can be explicitly converted to a pointer. The resulting pointer will have the same bit pattern as the original integral type.

9.1.2.8 Registers

Use of the register storage class in the declaration of an object has no effect on whether the object is placed in a register. The compiler performs register assignment aggressively; that is, it automatically attempts to place as many variables as possible into registers.

9.1.2.9 Classes, Structures, Unions, Enumerations, and Bit Fields

Accessing a member of a union by using a member of a different type results in an attempt to interpret, without conversion, the representation of the value of the member as the representation of a value in the different type.

Members of a class or structure are packed into Cray words from left to right. Padding is appended to a member to correctly align the following member, if necessary. Member alignment is based on the size of the member:

- For a member bitfield of any size, alignment is any bit position that allows the member to fit entirely within a 64–bit word.
- For a member with a size less than 64 bits, alignment is the same as the size. For example, a `char` has a size and alignment of 8 bits; a `float` or `short` on UNICOS/mk systems has a size and alignment of 32 bits.
- For a member with a size equal to or greater than 64 bits, alignment is 64 bits.
- For a member with array type, alignment is equal to the alignment of the element type.

A plain `int` type bit field is treated as an `unsigned int` bit field.

The values of an enumeration type are represented in the type `signed int` for C; they are a separate type in C++.

9.1.2.10 Qualifiers

When an object that has `volatile`-qualified type is accessed, it is simply a reference to the value of the object. If the value is not used, the reference need not result in a load of the value from memory.

9.1.2.11 Declarators

A maximum of 12 pointer, array, and/or function declarators are allowed to modify an arithmetic, structure, or union type.

9.1.2.12 Statements

The compiler has no fixed limit on the maximum number of case values allowed in a `switch` statement.

The C++ compiler parses `asm` statements for correct syntax, but otherwise ignores them.

9.1.2.13 Exceptions

In C++, when an exception is thrown, the memory for the temporary copy of the exception being thrown is allocated on the stack and a pointer to the allocated space is returned.

9.1.2.14 System Function Calls

See the exit(3) man page for a description of the form of the unsuccessful termination status that is returned from a call to exit.

9.1.3 Preprocessing

The value of a single-character constant in a constant expression that controls conditional inclusion matches the value of the same character in the execution character set. No such character constant has a negative value. For each, 'a' has the same value in the two contexts:

```
#if 'a' == 97
if ('a' == 97)
```

The -I option and the method for locating included source files is described in Section 2.20.1, page 32.

The source file character sequence in a #include directive must be a valid UNICOS file name or path name. A #include directive may specify a file name by means of a macro, provided the macro expands into a source file character sequence delimited by double quotes or < and > delimiters, as follows:

```
#define myheader "./myheader.h"
#include myheader

#define STDIO <stdio.h>
#include STDIO
```

The macros __DATE__ and __TIME__ contain the date and time of the beginning of translation. For more information, see the description of the predefined macros in Chapter 6, page 99.

The #pragma directives are described in section Chapter 4, page 61.

Libraries and Loaders [A]

This appendix describes the libraries that are available with the Cray C/C++ Programming Environment and the UNICOS loaders, `ld` and `cld`.

A.1 UNICOS Standard C and C++ Libraries

UNICOS libraries to support Cray Standard C and Cray C++ are automatically available on all Cray Research systems when you use the `CC`(1), `cc`(1), or `c89`(1) commands to compile your programs. These commands automatically issue the appropriate directives to load the program with the appropriate functions. If your program strictly conforms to the C or C++ standards, you do not need to know library names and locations. If your program requires other libraries or if you want direct control over the loading process, more knowledge of loaders and libraries is necessary.

A.2 UNICOS Loaders

When you issue the `cc`, `CC`, or `c89` commands to invoke the compiler and the program compiles without errors, the loader is called. Specifying the `-c` option on the command line produces relocatable object files without calling the loader. These relocatable object files can then be used as input to the loader command by specifying the file names on the appropriate loader command line.

For example, the following command line compiles a file called `target.c` and produces the relocatable object file called `target.o` in your current working directory.

```
cc -c target.c
```

You can then use file `target.o` as input to the loader or save the file to use with other relocatable object files to compile and create a linked executable file (`a.out` by default).

Because of the special code needed to handle templates, constructors, destructors, and other C++ language features, object files generated by using the `CC` command should be linked using the `CC` command. To link C++ object files using one of the loader commands (`ld` or `cld`), the `-h keep=`***files*** option (see Section 2.7.1, page 10) must be specified on the command line when compiling source files.

The UNICOS loaders, `ld` and `cld`, can be accessed by using one of the following methods:

- You can access the loader directly by using the `ld` or `cld` command. You can also use the `segldr` command to access `ld`.
- You can let the `cc`, `CC`, or `c89` command choose the loader. This method may cause slower loading and use more memory, but it also has the following advantages:
  - You do not need to know the loader command-line interface.
  - You do need to know which loader to call for the targeted machine.
  - You do not need to worry about the details of which libraries to load, or the order in which to load them.
  - In C++, you do not need to worry about template instantiation requirements or about loading the compiler-generated static constructors and destructors.

A.2.1 Loader for UNICOS Systems (SEGLDR)

The default loader on all UNICOS systems is SEGLDR. The `CC`, `cc`, and `c89` commands call SEGLDR by using the `ld`(1) command. Because SEGLDR was designed specifically for use with Cray language processors and UNICOS systems, it offers several advantages. Despite its name, SEGLDR produces both segmented and nonsegmented executable programs and is an efficient and full-featured loader for all types of programs. You can control SEGLDR operations with options on the `segldr` command line or directives in a directives file. For more information, see the `segldr`(1) man page in the *UNICOS User Commands Reference Manual*, and the *Segment Loader (SEGLDR) and ld Reference Manual*.

A.2.2 Loader for UNICOS/mk Systems (`cld`(1))

The default loader on UNICOS/mk systems is `cld`. The `CC`, `cc`, and `c89` commands call `cld` by using the `cld` command. Because `cld` was designed specifically for use with Cray language processors and UNICOS/mk systems, it offers several advantages. You can control `cld` operations with options on the `cld` command line or directives in a directives file. For more information, see the `cld`(1) man page.

Cray C/C++ Dialects [B]

This appendix details the features of the C and C++ languages that are accepted by the Cray Standard C and Cray C++ compilers, including certain language dialects and anachronisms. Users should be aware of these details, especially users who are porting codes from other environments.

B.1 C++ Conformance

The Cray C++ compiler accepts the C++ language as defined by *The Annotated C++ Reference Manual (ARM)* by Ellis and Stroustrup, Addison-Wesley, 1990, including templates, exceptions, and the anachronisms of Chapter 18. Many features have been updated to match the specification in the *X3J16/WG21 Working Paper*.

The Cray C++ compiler also has a `cfront` compatibility mode, which duplicates a number of features and bugs of `cfront` (Cray C++ 1.0 is based on `cfront`). Complete compatibility is not guaranteed or intended. The mode allows programmers who have used `cfront` features to continue to compile their existing code. Command-line options are also available to enable and disable anachronisms and strict standard-conformance checking. The command-line options are described in Chapter 2, page 3.

B.1.1 Supported Features

The following features, which are in the *X3J16/WG21 Working Paper* (but are not in *The Annotated C++ Reference Manual*), are supported:

- 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.
- A global-scope qualifier is allowed in member references of the form `x.::A::B` and `p->::A::B`.

- The precedence of the third operand of the ? operator is changed.
- If control reaches the end of the main() routine, and the main() routine has an integral return type, it is treated as if a return 0; statement was 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 that is 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 and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to const volatile cannot be bound to an rvalue.
- Qualification conversions such as conversion from T** to T const * const are allowed.
- Digraphs are recognized.
- Operator keywords (for example, and or bitand) are recognized.
- Static data member declarations can be used to declare member constants.
- wchar_t is recognized as a keyword and a distinct type.
- bool is recognized.
- RTTI (run-time type identification), including dynamic_cast and the typeid operator, is implemented.
- Declarations in tested conditions (within 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 a nested class outside its enclosing class is allowed.
- `mutable` is accepted on nonstatic 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 nonconverting 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 are retained on `rvalues` (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between process overlay directives (PODs) 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 can be used in an explicit destructor call.
- Placement delete is supported.
- An array allocated via a placement new can be deallocated via delete.
- `enum` types are considered to be nonintegral types.
- Partial specification of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as “guiding declarations” that are instances of the template.
- It is possible to overload operators using functions that take enum types and no class types.

- Explicit specification of function template arguments is supported.
- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.
- The notation `:: template` (and `-->template`, etc.) is supported.

B.1.2 Unsupported Features

The following features, which are in the *X3J16/WG21 Working Paper* (but are not in *The Annotated C++ Reference Manual*), are **not** supported:

- `extern inline` functions are not supported.
- Covariant return types on overriding virtual functions are not supported.
- `enum` types cannot contain values larger than can be contained in an `int` type.
- `reinterpret_cast` does not allow casting a pointer to a member of one class to a pointer to a member of another class if the classes are unrelated.
- Two-phase name binding in templates as described in the *Working Paper*, is not implemented.
- In a reference of the form `f()->g()`, with `g` being a static member function, `f()` is not evaluated. The *Working Paper* requires that `f()` be evaluated.
- Class name injection is not implemented.
- Putting a `try/catch` around the initializers and body of a constructor is not implemented.
- Template `template` parameters are not implemented.
- Koenig lookup of function names on all calls is not implemented.
- Finding friend functions of the argument class types on name lookup on the function name in calls is not implemented.
- String literals do not have `const` type.
- Universal character set escapes (e.g., `\uabcd`) are not implemented.
- The `export` keyword for templates is not implemented.

B.2 C++ Anachronisms Accepted

C++ anachronisms are enabled by using the `-h anachronisms` command-line option (see Section 2.4.4, page 7). When anachronisms are enabled, the following anachronisms are accepted:

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized by using the 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 can 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 can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is allowed. This is only allowed if anachronisms are enabled and the `assignment to this` configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a nonnested class name if no other class of that name has been declared. The anachronism is not applied to template classes.
- 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 can participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when checking for compatibility, therefore, the following statements declare 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.

B.3 Extensions Accepted in Normal C++ Mode

The following C++ extensions are accepted (except when strict standard conformance mode is enabled, in which case a warning or caution message may be issued):

- A `friend` declaration for a class can omit the `class` keyword, as shown in the following example:

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

- Constants of scalar type can be defined within classes, as shown in the following example:

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

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

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

- The restrict type qualifier for pointer, reference, and pointer-to-member types is allowed. See Section 3.1, page 43, for more information on restricted pointers.

- An assignment operator declared in a derived class with a parameter type matching one of its base classes is treated as a “default” assignment coperator; that is, such a declaration blocks the implicit generation of a copy

assignment operator. (This is cfront behavior that is known to be relied upon in at least one widely used library.) The following is an example:

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

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

- Implicit type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted. The following is an example:

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)()         // pf points to an extern "C++" function
             = &f;   // error unless implicit conversion allowed
```

B.4 Extensions Accepted in C or C++ Mode

The following extensions are accepted in C or C++ mode except when strict standard conformance modes is enabled, in which case a warning or caution message may be issued.

- The special lint comments /*ARGSUSED*/, /*VARARGS*/ (with or without a count of nonvarying arguments), and /*NOTREACHED*/ are recognized.
- A translation unit (input file) can contain no declarations.
- Comment text can appear at the ends of preprocessing directives.
- Bit fields can have base types that are enum or integral types in addition to int and unsigned int. This corresponds to A.6.5.8 in the ANSI Common Extensions appendix.
- Static functions can be declared in function and block scopes. Their declarations are moved to the file scope.
- enum tags can be incomplete as long as the tag name is defined and resolved by specifying the brace-enclosed list later.
- An extra comma is allowed at the end of an enum list.

- The final semicolon preceding the closing of a struct or union type specifier can be omitted.
- A label definition can be immediately followed by a right brace (}). (Normally, a statement must follow a label definition.)
- An empty declaration (a semicolon preceded by nothing) is allowed.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union does not need to be enclosed in braces. ANSI C requires braces.
- In an initializer, a pointer constant value can be cast to an integral type if the integral type is large enough to contain it.
- The address of a variable with register storage class may be taken.
- In an integral constant expression, an integer constant can be cast to a pointer type and then back to an integral type.
- In duplicate size and sign specifiers (for example, short short or unsigned unsigned) the redundancy is ignored.
- Benign redeclarations of typedef names are allowed. That is, a typedef name can be redeclared in the same scope with the same type.
- Dollar sign ($) and at sign (@) characters can be accepted in identifiers by using the -h calchars command-line option. This is not allowed by default.
- Numbers are scanned according to the syntax for numbers rather than the pp-number syntax. Thus, 0x123e+1 is scanned as three tokens instead of one token that is not valid. If the -h conform option is specified, the pp-number syntax is used.
- Assignment and pointer differences are allowed between pointers to types that are interchangeable but not identical, for example, unsigned char * and char *. This includes pointers to integral types of the same size (for example, int * and long *). Assignment of a string constant to a pointer to any kind of character is allowed without a warning.
- Assignment of pointer types is allowed in cases where the destination type has added type qualifiers that are not at the top level (for example, int ** to const int **). Comparisons and pointer difference of such pairs of pointer types are also allowed.

- In operations on pointers, a pointer to `void` is always implicitly converted to another type if necessary, and a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ANSI C, these are allowed by some operators, and not by others (generally, where it does not make sense).
- Pointers to different function types may be assigned or compared for equality (`==`) or inequality (`!=`) without an explicit type cast. This extension is not allowed in C++ mode.
- A pointer to `void` can be implicitly converted to or from a pointer to a function type.
- Restricted pointers are allowed. For more information, see Section 3.1, page 43.
- External entities declared in other scopes are visible:

  ```
  void f1(void) { extern void f(); }
  void f2() { f(); /* Using out of scope declaration */ }
  ```

- In C mode, end-of-line comments (`//`) are supported.
- The `long long` and `unsigned long long` types are accepted.
- Variable length arrays (VLAs) are supported in C mode.
- A non-lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

B.5 C++ Extensions Accepted in `cfront` Compatibility Mode

The `cfront` compatibility mode is enabled by the `-h cfront` command-line option (Cray C++ 1.0 is based on `cfront`). The following extensions are accepted in `cfront` compatibility mode:

- Type qualifiers on the `this` parameter are dropped in contexts such as in the following example:

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

  This is a safe operation. A pointer to a `const` function can be put into a pointer to non-`const`, because a call using the pointer is permitted to

modify the object and the function pointed to will not modify the object. The opposite assignment would not be safe.

- Conversion operators that specify a conversion to `void` are allowed.
- A nonstandard `friend` declaration can introduce a new type. A `friend` declaration that omits the elaborated type specifier is allowed in default mode, however, in `cfront` mode the declaration can also introduce a new type name. An example follows:

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

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

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

- A reference can be initialized to `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 0 is not considered to be a null pointer constant. In general, in overload resolution, a null pointer constant must be spelled “0” to be considered a null pointer constant (e.g., '\0' is not considered a null pointer constant).
- An alternate form of declaring pointer-to-member-function variables is supported, as shown in the following example:

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

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 the following single standard pointer-to-member declaration:

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

A nonstandard pointer-to-member declaration that appears outside of a class declaration, such as the declaration of `T`, is normally not valid and would cause an error to be issued. However, for declarations that appear within a class declaration, such as `A::T3`, this feature changes the meaning of a valid declaration. `cfront` version 2.1 accepts declarations, such as `T`, even when `A` is an incomplete type; so this case is also accepted.

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

```
class B { protected: int i; };
class D : public B { void mf()};

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

  Note: Protected member access checking for other operations (such as everything except taking a pointer-to-member address) is done normally.

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

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

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

  ```
  class A { A(); };
  double d;
  A x(int(d));
  A(x2);
  ```

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

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

  Similarly, the following declaration declares a function named `xzy`, that takes a parameter of type function taking no arguments and returning an `int`. In `cfront` mode, this is interpreted as a declaration of an object that is initialized with the value `int()`, which evaluates to 0.

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

- A named bit field can have a size of 0. The declaration is treated as though no name had been declared.

- Plain bit fields (such as bit fields declared with a type of `int`) are always unsigned.

- The name given in an elaborated type specifier can be a `typedef` name that is the synonym for a class name. For example:

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

- No warning is issued on duplicate size and sign specifiers, as shown in the following example:

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

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

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

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

- An extra comma is allowed after the last argument in an argument list. For example:

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

- A constant pointer-to-member function can be cast to a pointer-to-function, as in the following example. 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 like C structures, and the destructor is not called on the copy. In normal mode, the class object is copied into a temporary, the address of the temporary is passed as the argument, and the destructor is called on the temporary after the call returns. Because the argument is passed by value instead of by address, code like this compiled in `cfront` mode is not calling-sequence compatible with the same code

compiled in normal mode. In practice, this is not much of a problem, since classes that allow bitwise copying usually do not have destructors.

- A union member may be declared to have the type of a class for which the user has defined an assignment operator (as long as the class has no constructor or destructor). A warning is issued.
- When an unnamed class appears in a `typedef` declaration, the `typedef` name may appear as the class name in an elaborated type specifier. For example:

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

- Two member functions may be declared with the same parameter types when one is static and the other is nonstatic with a function qualifier. For example:

```
class A {
  void f(int) const;
  static void f(int); // No error in cfront mode
};
```

- The scope of a variable declared in the `for-init-statement` is the scope to which the `for` statement belongs. For example:

```
int f(int i) {
  for (int j = 0; j < i; ++j) { /* ... */ }
  return j; // No error in cfront mode
}
```

- Function types differing only in that one is declared `extern "C"` and the other `extern "C++"` can be treated as identical:

```
typedef void (*PF)();
extern "C" typedef void (*PCF)();
void f(PF);
void f(PCF);
```

  By contrast, in standard C++, PF and PCF are different and incompatible types; PF is a pointer to an `extern "C++"` function whereas PCF is a pointer to an `extern "C"` function; and the two declaractions of `f` create an overload set.

- Functions declared inline have internal linkage.

- `enum` types are regarded as integral types.
- An uninitialized const object of non-POD class type is allowed even if its default constructor is implicitly declared as in the following example:

```
struct A { virtual void f(); int i; };
const A a;
```

- A function parameter type is allowed to involve a pointer or reference to array of unknown bounds.

Compiler Messages [C]

This appendix describes how to use the message system to control and use messages issued by the compiler. Explanatory texts for messages can be displayed online through the use of the `explain`(1) command, described in the following section.

Messages and explanations are contained in message catalogs that can be locally modified and printed. If your use of a Cray system is normally through batch access, contact your site administrator to obtain a hard copy listing of messages and explanations.

For further information about the message system, see the *Cray Message System Programmer's Guide*, or the message system section of the *UNICOS System Libraries Reference Manual*. The introduction to that section can be viewed online as the `message`(3) man page.

C.1 Expanding Messages with the `explain` Command

If you are using a Cray Research system interactively, use the `explain`(1) command to display an explanation of any message issued by the compiler. The command takes as an argument the message number, including the number's prefix. The prefix for Cray Standard C and Cray C++ is `CC`.

In the following sample dialog, the `cc` command invokes the compiler on source file `bug.c`. Message `CC-24` is displayed. The `explain`(1) command displays the expanded explanation for this message.

```
> cc bug.c
CC-24 cc: ERROR File = bug.c, Line = 1
   An invalid octal constant is used.

   int i = 018;
             ^

1 error detected in the compilation of "bug.c".
> explain CC-24

An invalid octal constant is used.

Each digit of an octal constant must be between 0 and 7,
inclusive. One or more digits in the octal constant on the
indicated line are outside of this range. To avoid issuing
an error for each erroneous digit, the constant will be treated
as a decimal constant. Change each digit in the octal constant
to be within the valid range.
```

C.2 Controlling the Use of Messages

The following sections summarize the command-line options that affect the issuing of messages from the compiler.

C.2.1 Command-line Options

See the CC(1) man page for details about the following option descriptions.

Option	Description
`-h errorlimit[=`*n*`]`	Specifies the maximum number of error messages the compiler prints before it exits.
`-h [no]message=`*n*`[:...]`	Enables or disables the specified compiler messages, overriding `-h msglevel`.
`-h msglevel_`*n*	Specifies the lowest severity level of messages to be issued.

`-h report=`*args*	Generates optimization report messages.

C.2.2 Environment Options for Messages

The following environment variables are used by the message system. For more information, see the *Cray Message System Programmer's Guide*.

Variable	Description
NLSPATH	Specifies the default value of the message system search path environment variable.
LANG	Identifies your requirements for native language, local customs, and coded character set with regard to the message system.
MSG_FORMAT	Controls the format in which you receive error messages.

C.2.3 ORIG_CMD_NAME Environment Variable

You can override the command name printed in the message. If the environment variable ORIG_CMD_NAME is set, the value of ORIG_CMD_NAME is used as the command name in the message. This functionality is provided for use with shell scripts that invoke the compiler. By setting ORIG_CMD_NAME to the name of the script, any message printed by the compiler appears as though it was generated by the script. For example, the following C shell script is named newcc:

```
#
setenv ORIG_CMD_NAME 'basename $0'
cc $*
```

A message generated by invoking newcc resembles the following:

```
CC-8 newcc: ERROR File = x.c, Line = 1
  A new-line character appears inside a string literal.
```

Because the environment variable ORIG_CMD_NAME is set to newcc, this appears as the command name instead of cc in this message.

Caution: The ORIG_CMD_NAME environment variable is not part of the message system. It is supported by the C/C++ compiler as an aid to programmers. Other products, such as the Fortran compiler and the loader, may support this variable. However, you should not rely on support for this variable in any other product.

You must be careful when setting the environment variable ORIG_CMD_NAME. If you set ORIG_CMD_NAME inadvertently, the compiler may generate messages with an incorrect command name. This may be particularly confusing if, for example, ORIG_CMD_NAME is set to newcc when the CF90 compiler prints a message. The Fortran message will look as though it came from newcc.

C.3 Message Severity

Each message issued by the compiler falls into one of the following categories of messages, depending on the severity of the error condition encountered or the type of information being reported.

Category	Meaning
COMMENT	Inefficient programming practices.
NOTE	Unusual programming style or the use of outmoded statements.
CAUTION	Possible user error. Cautions are issued when the compiler detects a condition that may cause the program to abort or behave unpredictably.
WARNING	Probable user error. Indicates that the program will probably abort or behave unpredictably.
ERROR	Fatal error; that is, a serious error in the source code. No binary output is produced.
INTERNAL	Problems in the compilation process. Please report internal errors immediately to the system support staff, so that the problem can be forwarded to CRI Software Development for resolution.
LIMIT	Compiler limits have been exceeded. Normally you can modify the source code or environment to avoid these errors. If limit errors cannot be resolved by such modifications, please report

these errors to the system support staff, so that the problem can be forwarded to CRI Software Development for resolution.

`INFO`	Useful additional information about the compiled program.
`TASKING`	Information about tasking optimizations performed on the compiled code.
`INLINE`	Information about inline code expansion performed on the compiled code.
`SCALAR`	Information about scalar optimizations performed on the compiled code.
`VECTOR`	Information about vectorization optimizations performed on the compiled code.

C.4 Common System Messages

The four errors in the following list can occur during the execution of a user program. The operating system detects them and issues the appropriate message. These errors are not detected by the compiler and are not unique to C or C++; they may occur in any application program written in any language.

- `Operand Range Error`

  An operand range error occurs when a program attempts to load or store in an area of memory that is not part of the user's area. This usually occurs when an invalid pointer is dereferenced.

- `Program Range Error`

  A program range error occurs when a program attempts to jump into an area of memory that is not part of the user's area. This may occur, for example, when a function in the program mistakenly overwrites the internal program stack. When this happens, the address of the function from which the function was called is lost. When the function attempts to return to the calling function, it jumps elsewhere instead.

- `Error Exit`

  An error exit occurs when a program attempts to execute an invalid instruction. This error usually occurs when the program's code area has

been mistakenly overwritten with words of data (for example, when the program stores in a location pointed to by an invalid pointer).

- `Floating-point Exception`

  A floating-point exception occurs when a program attempts to perform a floating-point operation that is not valid. On UNICOS systems, this error can occur in integer arithmetic because some integer operations are performed with floating-point arithmetic.

Intrinsic Functions [D]

The Cray C/C++ intrinsic functions either allow for direct access to some Cray Research hardware instructions or result in generation of inline code to perform some specialized functions. These intrinsic functions are processed completely by the compiler. In many cases, the generated code is one or two instructions. These are called *functions* because they are invoked with the syntax of function calls.

To get access to the intrinsic functions, the Cray C++ compiler requires that either the `intrinsics.h` file be included or that the intrinsic functions that you want to call be explicitly declared. If you explicitly declare an intrinsic function, the declaration must agree with the documentation or the compiler treats the call as a call to a normal function, not the intrinsic function. When using the Cray Standard C compiler, it is not necessary to declare intrinsic functions. In either case, the `-h nointrinsics` command-line option causes the compiler to treat these calls as regular function calls and not as intrinsic function calls.

The types of the arguments to intrinsic functions are checked by the compiler, and if any of the arguments do not have the correct type, a warning message is issued and the call is treated as a normal call to an external function. If your intention was to call an external function with the same name as an intrinsic function, you should change the external function name. The names used for the Cray Standard C intrinsic functions are in the name space reserved for the implementation.

For detailed descriptions of appropriate Cray Research hardware instructions, see the *Symbolic Machine Instructions Reference Manual*.

Note: Several of these intrinsic functions have both a vector and a scalar version on UNICOS systems. If a vector version of an intrinsic function exists and the intrinsic is called within a vectorized loop, the compiler uses the vector version of the intrinsic. See the appropriate intrinsic function man page for details on whether it has a vector version.

The following table provides a summary of all C/C++ intrinsic functions and indicates their availability on Cray Research computer systems. See the appropriate man page for more information.

Table 4. Summary of C/C++ intrinsic functions

Function	UNICOS systems	UNICOS/mk systems
`_argcount` (See Footnote 1)	X	X
`_cmr`	X	
`_dshiftl`	X	X
`_dshiftr`	X	X
`_EX`	X	
`_gbit`	X	X
`_gbits`	X	X
`_getvm`	X	
`_int_mult_upper` (See Footnote 2)	X	X
`_leadz`	X	X
`_mask`	X	X
`_maskl`	X	X
`_maskr`	X	X
`_mclr` (See Footnote 3)	X	X
`_mld` (See Footnote 3)	X	X
`_memory_barrier`		X
`_mldmx` (See Footnote 3)	X	X
`_mmx` (See Footnote 3)	X	X
`_mul` (See Footnote 3)	X	X
`_my_pe`		X
`_numargs`	X	X
`_num_pes`		X

Function	UNICOS systems	UNICOS/mk systems
_pbit	X	X
_pbits	X	X
_popcnt	X	X
_poppar	X	X
_ranf	X	X
_readSB	X	
_readSR	X	
_readST	X	
_remote_write_barrier		X
_rtc	X	X
_semclr	X	
_semget	X	
_semput	X	
_semset	X	
_semts	X	
_setvm	X	
_write_memory_barrier		X
_writeSB	X	
_writeST	X	

Footnote 1: Available only on CRAY T90 systems.
Footnote 2: Available only on Cray Research systems with IEEE floating-point hardware.
Footnote 3: This function is not available on some CRAY Y-MP systems.

Index

-#, 31
-##, 31
-###, 31
// comments, 48

A

align directive, 81
Anachronisms
  C++, 145
Analysis tools
  -F option, 25
  -h [no]apprentice, 26
  -h [no]atexpert, 26
  -h [no]listing, 26
Apprentice, 26
_argcount, 164
Argument passing, 109
Arithmetic
  See "Math", 23, 24
Arithmetic conversion, 50
Arithmetic conversions, 59
Array storage, 109
Arrays, 135
  dependencies, 83
  variable length, 50
    declarator restrictions, 50
    declarators, 51
    function declarators, 52
    goto statement, 55
    longjmp function, 56
    setjmp function, 56
    sizeof operator, 54
    switch statement, 55
    type definitions, 53
asm statements, 137
Assembly language
  output, 30
Assembly language functions, 107
Assembly source expansions, 3
ATExpert, 26
Automatic instantiation, 92
Autotasking, 17, 18, 71
  level, 16

B

Barrier/eureka synchronization units, 63
besu directive, 63
Bit fields, 136
Blank common block, 114
Block scope, 45
Bottom loading, 20
bounds directive, 64
Branches
  vs. jumps, 22
btol conversion utility, 112

C

C extensions, 43
  conversions of complex type, 59
  variable length arrays, 50
-c, 139
-C option, 34
-c option, 30
cache_align directive, 81
cache_bypass directive, 82
Calls, 107
can_instantiate directive, 68, 96
case directive, 73
cfront, 149
  compatibility mode, 141
cfront compilers, 7

Character data, 111
Character pointers, 135
Character set, 130
Characters
  wide, 131
CIV
  See "Constant increment variables", 21
Classes, 136
_cmr, 164
Command-line options
  -# option, 31
  -## option, 31
  -### option, 31
  -C option, 34
  -c option, 3, 30
  compiler version, 37
  conflicting with directives, 6
  conflicting with other options, 6
  -D macro[=def], 33
  -d string option, 37
  defaults, 5
  -E option, 3, 30
  examples, 39
  -F option, 25, 37
  -G option, 26, 103, 104
  -g option, 26, 103, 104
  -h [no]abort, 29
  -h [no]aggress, 13
  -h [no]align, 20
  -h [no]anachronisms, 7
  -h [no]apprentice, 26
  -h [no]atexpert, 26
  -h [no]autoinstantiate, 8
  -h [no]bl, 20
  -h [no]bounds, 27, 103
  -h [no]calchars, 11
  -h [no]conform, 6
  -h [no]exceptions, 7
  -h [no]fastfpdivide, 24
  -h [no]fastmd, 23
  -h [no]fastmodulus, 23
  -h [no]ieeeconform, 24
  -h [no]implicitinclude, 9
  -h [no]interchange, 19
  -h [no]intrinsics, 13
  -h [no]ivdep, 15
  -h [no]jump, 22
  -h [no]listing, 26
  -h [no]message=n, 28
  -h [no]overindex, 13
  -h [no]pattern, 13
  -h [no]pragma=name[:name...], 34
  -h [no]reduction, 21
  -h [no]rounddiv, 25
  -h [no]signedshifts, 11
  -h [no]split, 22
  -h [no]stack, 12
  -h [no]taskinner, 17
  -h [no]threshold, 18
  -h [no]tolerant, 8
  -h [no]trunc, 103
  -h [no]trunc[=n], 25
  -h [no]unroll, 22
  -h [no]vsearch, 16
  -h [no]zeroinc, 21
  -h anachronisms, 145
  -h cfront, 7, 149
  -h common, 17
  -h errorlimit[=n], 29
  -h feonly, 30
  -h forcevtble, 9
  -h ident=name, 39
  -h indef, 27
  -h inlinefrom=file, 19
  -h inlinen, 18
  -h instantiate=mode, 8
  -h keep=file, 10
  -h matherror=method, 23
  -h msglevel_n, 28
  -h new_for_init, 7
  -h options
    errorlimit, 158
  -h pipelinen, 21
  -h prelink_local_copy, 9
  -h remove_instantiation_flags, 9

-h report=args, 28
-h restrict=args, 10
-h scalarn, 19
-h suppressvtble, 9
-h taskcommon, 17
-h taskn, 16
-h taskprivate, 17
-h vectorn, 15
-h zero, 27
-I option, 32
-L libdir option, 36
-l libfile option, 35
-M option, 34
macro definition, 33
-N option, 34
-O level, 12
-o option, 37
-P option, 3, 30
prelink_copy_if_nonlocal, 9
preprocessor options, 30
remove macro definition, 34
-S option, 3, 30
-s option, 37
-U macro option, 34
-V option, 37
-W option, 31
-X npes option, 38
-Y option, 32
Commands
c89, 1, 3
files, 4
format, 4
CC, 1, 3
files, 4
format, 4
cc, 1, 3
files, 4
format, 4
compiler, 3
cpp, 3
format, 5
ld, 10
options, 5
Comments
in code, 48
preprocessed, 34
Common block, 114
common directive, 77
Common system messages, 161
Compilation phases
-#, 31
-##, 31
-###, 31
-c option, 30
-E option, 30
-h feonly, 30
-P option, 30
-S option, 30
-Wphase["opt..."], 31
-Yphase,dirname, 32
Compiler
Cray C++, 2
Cray Standard C, 2
Compiler messages, 157
Complex
adding types, 59
arithmetic conversion, 50
conversion to and from, 49
data types, 48
usage, 49
value
printing, 49
variable, 49
complex types, 50
concurrent directive, 83
Conformance
C++, 141
Constant increment variables (CIVs), 21
Constructs
accepted and rejected, 7
old, 8
Conversion utility
_btol, 112
_cptofcd, 111
_fcdlen, 111

_fcdtocp, 111
_ltob, 112
_cptofcd conversion utility, 111
Cray Assembler for MPP (CAM), 108
Cray Assembly Language (CAL), 108
Cray C++ Compiler, 2
Cray Standard C Compiler, 2
Cray SV1 system, 79
Cray TotalView debugger, 103
CRI_c89_OPTIONS, 41
CRI_CC_OPTIONS, 41
CRI_cc_OPTIONS, 41
CRI_cpp_OPTIONS, 41

D

-D macro[=def], 33
-d string, 37
Data types, 128
complex, 48
long long, 47
mapping (table), , 129
unsigned long long, 47
__DATE__ , 138
Debugger, 103
Debugging, 26
features, 103
-G level, 26
-g option, 26
-h [no]bounds, 27
-h indef, 27
-h zero, 27
Debugging options, 104
Declarators, 137
function, 52
restrictions, 50
VLA, 51
Declared bounds, 13
Dialects, 141
Directives
arguments to tasking, 78
C++, 62
conflicts with options, 6
#define, 33
diagnostic messages, 62
disabling, 35
general, 63
#include, 32
inlining, 88
instantiation, 68
loop, 62
macro expansion, 61
MSP, 80
examples, 80
#pragma , 61
align, 81
alternative form, 63
arguments to instantiate, 96
besu, 63
cache_align, 81
cache_bypass, 82
can_instantiate, 68, 96
case, 73
common, 77
concurrent, 83
do_not_instantiate, 68, 96
duplicate, 65
endcase, 73
endguard, 74
endloop, 73
endparallel, 72
format, 61
guard, 74
ident, 68
in C++, 62
inline, 89
instantiate, 68, 95
ivdep, 46, 69
message, 66, 103
[no]bounds, 64
[no]bounds directive, 103
[no]opt, 66, 103
noinline, 89
nointerchange, 83

noreduction, 84
nostream, 80
novector, 70
novsearch, 70
parallel, 72
preferstream, 80
prefertask, 77
prefervector, 70
shortloop, 71
shortloop128, 71
soft, 67
split, 84
suppress, 85
symmetric, 86
taskcommon, 76
taskloop, 72
taskprivate, 74
taskshared, 75
unroll, 87
usage, 61
uses_eregs, 66
vfunction, 68
preprocessing, 138
protecting, 62
scalar, 81
tasking, 71
vectorization, 69
work distribution, 78
Directories
#include files, 32
library files, 35, 36
phase execution, 32
do_not_instantiate directive, 68, 96
double, 132
double complex, 48
_dshiftl, 164
_dshiftr, 164
duplicate directive, 65

E

E register, 83
E registers, 66
-E option, 30
E-registers
cache_bypass, 82
endcase directive, 73
endguard directive, 74
endloop directive, 73
endparallel directive, 72
–h interchange, 19
–h nointerchange, 19
Enumerations, 136
Environment, 127
Environment variables, 41
CRI_c89_OPTIONS, 41
CRI_CC_OPTIONS, 41
CRI_cc_OPTIONS, 41
CRI_cpp_OPTIONS, 41
LANG, 41, 159
MSG_FORMAT, 41, 159
NLSPATH, 41, 159
NPROC, 41
ORIG_CMD_NAME, 159
TARGET, 41
Epsilon value, 133, 134
Error Exit, 161
Error messages, 157
_EX, 164
Examples
command-line, 39
Exception construct, 7
Exception handling, 7
Exceptions, 138
explain, 157
Extensions, 43
C++ mode, 146
cfront compatibility mode, 149
#pragma directives, 61
extern "C" keyword, 123
External functions
declaring, 107

F

-F option, 25
_fcdlen conversion utility, 111
_fcdtocp conversion utility, 111
Features
  C++, 141
    cfront compatibility, 141
Files
  a.out, 3
  compiler information file (CIF), 26
  constructor/destructor, 10
  default library, 35
  dependencies, 34
  .ii file, 93
  inlining, 19
  intrinsics.h, 163
  library directory, 36
  linking, 10
  listing, 26
  output, 37
  personal libraries, 36
float, 132
float complex, 48
Floating constants, 58
Floating-point
  arithmetic, 132
  overflow, 132
Floating-point arithmetic
  IEEE, 133
  rounding, 133, 134
Floating-point constants, 58
Floating-point Exception, 162
Flowtrace, 25
Fortran common block, 114
fortran keyword, 57
fortran.h header, 111
friend declaration, 150
Function parameters, 44
Functions, 163
  and variable length arrays, 52
  mbtowc, 131

G

-G level, 26
-G option, 103, 104
-g option, 103, 104
_gbit, 164
_gbits, 164
General command functions
  -h ident=name, 39
  -V option, 37
  -Xnpes, 38
_getvm, 164
goto statement, 55
guard directive, 74
Guarded region, 74
Guards
  numbered, 74
  unnumbered, 74

H

-h [no]implicitinclude, 9
-h [no]message=n[:...], 158
-h [no]message=n[:n...], 28
-h [no]pragma=name[:name...], 34
-h abort, 29
-h aggress, 13
-h align, 20
-h anachronisms, 7, 145
-h apprentice, 26
-h atexpert, 26
-h autoinstantiate, 8
-h bl, 20
-h bounds, 27, 103
-h calchars, 11
-h cfront, 7
-h common, 17
-h conform, 6
-h errorlimit, 158
-h errorlimit[=n], 29, 158
-h exceptions, 7

-h fastfpdivide, 24
-h fastmd, 23
-h fastmodulus, 23
-h feonly, 30
-h forcevtbl, 9
-h ident=name, 39
-h ieeeconform, 24
-h implicitinclude, 9
-h indef, 27
-h inlinefrom=file, 19
-h inlinen, 18
-h instantiate=mode, 8
-h intrinsics, 13
-h ivdep, 15
-h jump, 22
-h keep=file, 10
-h listing, 26
-h matherror=method, 23
-h msglevel_n, 28, 158
-h new_for_init, 7
-h noabort, 29
-h noaggress, 13
-h noalign, 20
-h noanachronisms, 7
-h noapprentice, 26
-h noatexpert, 26
-h noautoinstantiate, 8
-h nobl, 20
-h nobounds, 27, 103
-h nocalchars, 11
-h noconform, 6
-h noexceptions, 7
-h nofastfpdivide, 24
-h nofastmd, 23
-h nofastmodulus, 23
-h noieeeconform, 24
-h nointrinsics, 13, 163
-h noivdep, 15
-h nojump, 22
-h nolisting, 26
-h nooverindex, 13
-h nopattern, 13
-h noreduction, 21
-h norounddiv, 25
-h nosearch, 16
-h nosignedshifts, 11
-h nosplitn, 22
-h nostack, 12
-h notaskinner, 17
-h nothreshold, 18
-h notolerant, 8
-h notrunc, 103
-h notrunc[=n], 25
-h nounroll, 22
-h nozeroincn, 21
-h overindex, 13
-h pattern, 13
-h pipelinen, 21
-h prelink_copy_if_nonlocal, 9
-h prelink_local_copy, 9
-h reduction, 21
-h remove_instantiation_flags, 9
-h report=args, 28, 159
-h restrict=args, 10
-h rounddiv, 25
-h scalarn, 19
-h search, 16
-h signedshifts, 11
-h splitn, 22
-h stack, 12
-h streamn, 79
-h suppressvtbl, 9
-h taskcommon, 17
-h taskinner, 17
-h taskn, 16
-h taskprivate, 17
-h threshold, 18
-h tolerant, 8
-h trunc, 103
-h trunc[=n], 25
-h unroll, 22
-h vectorn, 15
-h zero, 27
-h zeroincn, 21
Hardware

intrinsic functions, 13
Hexadecimal floating constant, 58

I

-I incldir, 32
ident directive, 68
Identifier names
allowable, 11
Identifiers, 128
IEEE floating-point representation, 133
IEEE floating-point standard conformance, 24
Implementation-defined behavior, 127
Implicit inclusion, 9, 97
inline directive, 89
Inlining, 88
level, 18
instantiate directive, 68, 95
Instantiation
enable or disable, 8
local files, 9
modes, 8
nonlocal object file recompiled, 9
remove flags, 9
template, 91
Instantiation directives, 68, 95
Instantiation modes
all, 95
local, 95
none, 95
used, 95
_int_mult_upper, 164
Integers
overflow, 132
representation, 131
Interchange loops, 19
Interlanguage communication, 107
argument passing, 109
array storage, 109
assembly language functions, 107
blank common block, 114
CAL functions, 108
calling a C program from C++, 123
calling a C/C++ function from Fortran, 120
calling a Fortran program from C++, 119
calling Fortran routines, 108
CAM functions, 108
logical and character data, 111
Intermediate translations, 3
Intrinsic function
alog, 69
cos, 69
coss, 69
exp, 69
_popcnt, 69
pow, 69
ranf, 69
sin, 69
sqrt, 69
Intrinsic functions
argument types, 163
summary, 163
Intrinsics, 13
intrinsics.h, 163
ivdep, 82
ivdep directive, 69

J

Jumps
vs. branches, 22

K

K & R preprocessing, 34
Keywords
extern "C", 123
fortran, 57
restrict, 43

L

-L libdir, 36
-l libfile, 35
LANG, 41, 159
Language
  general
    -h [no]calchars, 11
    -h [no]calsignedshifts, 11
    -h [no]stack, 12
    -h keep=file, 10
    -h restrict=args, 10
  standard conformance
    -h [no]anachronisms, 7
    -h [no]conform, 6
    -h [no]exceptions, 7
    -h [no]tolerant, 8
    -h cfront, 7
    -h new_for_init, 7
  templates
    -h [no]autoinstantiate, 8
    -h [no]implicitude, 9
    -h instantiate=mode, 8
    -h prelink_copy_if_nonlocal, 9
    -h prelink_local_copy, 9
    -h remove_instantiation_flags, 9
  virtual functions
    -h forcevtbl, 9
    -h suppressvtbl, 9
_leadz, 164
Libraries
  default, 36
  Standard C, 139
Library, Standard Template, 2
Limits, 127
  implementation, 46
Linking
  files, 10
Loader
  cld, 140
  -d string, 37
  default, 139
  -L libdir, 36
  -l libfile, 35
  -o outfile, 37
  -s option, 37
  segldr, 140
Loaders
  -## option, 37
  ld, 3, 37
  mppld, 3, 37
Local memory references, 82
Logical data, 111
long double, 132
long double complex, 48
long long, 59
long long data types, 47
longjmp function, 56
Loop directives, 62
Loop splitting, 84
Loop unrolling, 87
Loops
  split, 22
  unrolling, 22
_ltob conversion utility, 112

M

-M option, 34
Macros, 108
  expansion in directives, 61
  removing definition, 34
Macros, predefined, 99
  _ADDR32, 101
  _ADDR64, 101
  __cplusplus, 100
  CRAY, 100
  cray, 100
  CRAY1, 100
  _CRAY1, 101
  _CRAY, 100
  _CRAYC, 102
  _CRAYIEEE, 101
  _CRAYMPP, 101

_CRAYT3E, 101
__DATE__, 99
_FASTMD, 101
__FILE__, 99
_LD64, 101
__LINE__, 99
_MAXVL, 101
_RELEASE, 102
__STDC__, 99
__TIME__, 99
_UNICOS, 100
unix, 100
__unix, 100
_mask, 164
_maskl, 164
_maskr, 164
Math
functions
arithmetic conversions, 50
double complex arguments, 50
-h [no]fastfpdivide, 24
-h [no]fastmd, 23
-h [no]fastmodulus, 23
-h [no]ieeeconform, 24
-h [no]rounddiv, 25
-h [no]trunc[=n], 25
-h matherror=method, 23
mbtowc, 131
_mclr, 164
_memory_barrier, 164
message directive, 66, 103
Messages, 127, 157
common system, 161
Error Exit, 161
Floating-point Exception, 162
Operand Range Error, 161
Program Range Error, 161
for _CRI directives, 62
-h [no]abort, 29
-h [no]message=n[:n...], 28
-h errorlimit[=n], 29
-h msglevel_n, 28
-h report=args, 28
option summary, 158
severity, 160
CAUTION, 160
COMMENT, 160
ERROR, 160
INFO, 161
INLINE, 161
INTERNAL, 160
LIMIT, 160
NOTE, 160
SCALAR, 161
TASKING, 161
VECTOR, 161
WARNING, 160
Microtasking, 72
_mld, 164
_mldmx, 164
_mmx, 164
modules, 1
MSG_FORMAT, 41, 159
MSP, 79
directives, 80
-h streamn, 79
sysconf command, 79
_mul, 164
Multi-streaming processor
See "MSP", 79
_my_pe, 164

N

-N option, 34
Names, 128
NLSPATH, 41, 159
nobounds directive, 64
noinline directive, 89
nointerchange directive, 83
noopt directive, 66, 103
noreduction directive, 84
nostream directive, 80
novector directive, 70

novsearch directive, 70
NPROC, 41
_num_pes, 164
_numargs, 164
Numbered guards, 74

O

-o outfile, 37
-O[level], 12
Operand Range Error, 161
Operators
  bitwise and integers, 131
opt directive, 66, 103
Optimization
  automatic scalar, 19
  general
    -h [no]aggress, 13
    -h [no]intrinsics, 13
    -h [no]overindex, 13
    -h [no]pattern, 13
    -O level, 12
  inline
    -h inlinefrom=file, 19
    -h inlinen, 18
  interchange loops, 19
  limitations, 13
  MSP, 79
  scalar
    -h [no]align, 20
    -h [no]bl, 20
    -h [no]interchange, 19
    -h [no]reduction, 21
    -h scalarn, 19
  task
    -h [no]taskinner, 17
    -h [no]threshold, 18
    -h taskcommon, 17
    -h taskn, 16
    -h taskprivate, 17
  UNICOS/mk specific
    -h [no]jumpn, 22
    -h [no]splitn, 22
    -h [no]unrolln, 22
    -h pipelinen, 21
  vector
    -h [no]ivdep, 15
    -h [no]vsearchn, 16
    -h [no]zeroincn, 21
    -h vectorn, 15
Optimization level, 12
Options
  conflicts, 6
  See "Command-line options", 6
ORIG_CMD_NAME, 159
Overindexing, 13

P

-P option, 30
parallel directive, 72
Parallel processing
  tasking directives, 71
Parallel region, 72
Pattern matching
  enable or disable, 13
_pbit, 165
_pbits, 165
Performance
  improvement, 15
Pipelining
  levels, 21
Pointer parameters, 44
Pointers, 135
  function parameter, 10
  restricted, 10
    See "Restricted pointers", 43
  this, 10
  UNICOS systems, 135
  UNICOS/mk systems, 136
_popcnt, 165
_poppar, 165
Porting code, 8, 141

#pragma directives
See "Directives", 61
_Pragma directives, 63
Predefined macros, 99
preferstream directive, 80
prefertask directive, 77
prefervector directive, 70
Prelinker, 92
Preprocessing, 138
-C option, 34
-D macro[=def], 33
-h [no]pragma=name[:name...] , 34
-I incldir, 32
-M, 34
-N option, 34
old style (K & R), 34
retain comments, 34
-U macro, 34
Preprocessor, 30
passing arguments to, 31
Preprocessor phase, 3
Processing elements (PEs), 38
Program Range Error, 161
Programming environment
description, 1
setup, 1
Protected member access checking, 151

Q

Qualifiers, 137

R

_ranf, 165
_readSB, 165
_readSR, 165
_readST, 165
Reduction loop, 84
Reduction loops, 21
Registers, 136
Relocatable object file, 30
Relocatable object files, 3
_remote_write_barrier, 165
Restricted pointers, 10, 43
block scope, 45
compared to ivdep, 46
file scope, 45
function parameters, 44
implementation limits, 46
unrestricted, 46
Rounding, 25
floating-point, 134
_rtc, 165

S

-S option, 30
-s option, 37
Scalar directives, 81
Search
library files, 36
Search loops, 16
SEGLDR, 139
accessing, 140
_semclr, 165
_semget, 165
_semput, 165
_semset, 165
_semts, 165
setjmp function, 56
Setting up environment, 1
_setvm, 165
Shift operator, 132
shortloop directive, 71
shortloop128 directive, 71
sizeof, 128
sizeof operator, 54
soft directive, 67
Soft externals, 67
split directive, 84
Standard Template Library, 2

Standards, 127
  arrays and pointers, 135
  bit fields, 136
  C violation, 8
  character set, 130
    example, 130
  classes, 136
  conformance to, 6
  Cray floating-point representation, 132
  data types, 128
    mapping, , 129
  declarators, 137
  enumerations, 136
  environment, 127
  exceptions, 138
  extensions, 43
  floating-point arithmetic, 132
    double, 132
    epsilon value, 133
    float, 132
    long double, 132
    maximum positive value, 132
  identifiers, 128
  IEEE floating-point representation, 133
    maximum positive value, 134
  implementation-defined behavior, 127
  integers, 131
  messages, 127
  pointers
    UNICOS systems, 135
    UNICOS/mk systems, 136
  preprocessing, 138
  qualifiers, 137
  register storage class, 136
  statements, 137
  structures, 136
  system function calls, 138
  unions, 136
  wide characters, 131
Statements, 137
STL
  See "Standard Template Library", 2
Storage class, 57
Stream buffer, 83
Stripmining, 85
Structures, 136
suppress directive, 85
switch statement, 55
Symbolic information, 37
symmetric directive, 86
Syntax checking, 30
sysconf command, 79
System function calls, 138

T

TARGET, 41
taskcommon directive, 76
Tasking
  autotasking, 71
  context arguments, 78
  directive arguments, 78
  miscellaneous arguments, 79
  status, 17
  user-directed, 72
Tasking level, 16
taskloop directive, 72
taskprivate directive, 74
taskshared directive, 75
Template, 91
Template instantiation, 91
  automatic, 92
  directives, 95
  implicit inclusion, 97
  modes, 94
Throw expression, 7
Throw specification, 7
__TIME__, 138
TotalView debugger, 104
Try block, 7
Types, 128

U

-U macro, 34
UNICOS
 C libraries, 139
 loader, 139
UNICOS message system, 157
Unions, 136
Unnumbered guards, 74
Unrestricted pointers, 46
unroll directive, 87
unsigned long long data types, 47
uses_eregs directive, 66

V

-V option, 37
Variable
 complex, 49
Variable length arrays
 See "Arrays", 50
Variables, 41
 context arguments, 78
Vectorization
 automatic, 15
 dependency analysis, 15
 directives, 69
 level, 15
 search loops, 16
vfunction directive, 68
Virtual function table, 9
VLA
 See "Arrays", 50
volatile qualifier, 86

W

Work distribution, 78
-Wphase["opt..."], 31
_write_memory_barrier, 165
_writeSB, 165
_writeST, 165

X

-X npes, 38

Y

-Yphase,dirname, 32