Cray C/C++ Reference Manual

SR–2179 2.0

Cray Research, Inc.

Portions of this product may still be in development. The existence of those portions still in development is not a commitment of actual release or support by Cray Research, Inc. Cray Research, Inc. assumes no liability for any damages resulting from attempts to use any functionality or documentation not officially released and supported. If it is released, the final form and the time of official release and start of support is at the discretion of Cray Research, Inc.

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.

Cray Research Software Documentation Map

The information on the following pages highlights the major body of documentation available for Cray Research customers. The manuals are organized into categories by product designation:

Product	Description
Programming environments	Cray Research programming environments provide an integrated development environment by combining compilers with development tools and libraries.
UNICOS operating system	The UNICOS operating system provides basic system control and management functions. It provides services, including basic batch and interactive system access for users, operating system interfaces for programmers, networking facilities, and sophisticated resource management for administrators. UNICOS MAX, in conjunction with UNICOS on the host system, provides operating system functionality on CRAY T3D systems.
Networking products	Cray Research networking products provide media connections, network programming access, and batch and interactive access for users across a heterogeneous network of computers.
Operating systems supplementary products	The operating systems supplementary products include a variety of products that are used in conjunction with Cray Research operating systems but are not included on the same media.

To use the map, find the product designation closest to your specific needs or role as a Cray Research system user. Documents listed under other categories also may be of interest to you; documents are listed only once, underneath the product to which they most directly apply. The date on the bottom of the page tells you when the information was last revised.

For more information

In addition to the map, you can use the following publications to find documentation specific to your needs:

- *Software Documentation Ready Reference,* publication SQ–2122, serves as a general index to the Cray Research documentation set. The booklet lists documents and man pages according to topic.
- *Software Overview for Users*, publication SG–2052, provides an overview of Cray Research software, including operating systems, programming environments, libraries, tools, applications, and so on. It directs you to documentation that contains user-level information.
- *User Publications Catalog*, publication CP–0099, briefly describes all Cray Research manuals available to you, including some not shown on the map. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on CRInform.

Ordering

To obtain Cray Research publications, order them by publication number from the Distribution Center:

Cray Research, Inc.
Distribution Center
2360 Pilot Knob Road
Mendota Heights, MN 55120
USA

Order desk	+1–612–683–5907
Fax number	+1–612–452–0141
Email	`orderdsk@cray.com`

Programming Environments

Ada

Cray Ada Environment, Volume 1: Reference Manual (SR–3014)

Cray Ada Environment, Volume 2: Programming Guide (SR–3082)

Assemblers

Cray Assembler for MPP (CAM) Reference Manual (SR–2510)†

Cray Assembly Language (CAL) for Cray PVP Systems Ready Reference (SQ–3110)

Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual (SR–3108)†

Fortran Compilers

CF90 Ready Reference (SQ–3900)

CF90 Commands and Directives Reference Manual (SR–3901)†

CF90 Fortran Language Reference Manual, Vol. 1 (SR–3902)†

CF90 Fortran Language Reference Manual, Vol. 2 (SR–3903)†

CF90 Fortran Language Reference Manual, Vol. 3 (SR–3905)†

CF90 SPARC Ready Reference (SQ–3904)

Introducing the CF90 SPARC Programming Environment (IN–2155)†

Cray MPP Fortran Reference Manual (SR–2504)†

CF77 Fortran Language Reference Manual (SR–3772)

Cray Standard C Compiler

Cray Standard C Reference Manual (SR–2074)†

Cray Standard C Ready Reference (SQ–2076)

Cray Standard C Reference Manual for MPP (SR–2506)†

C++ Compiler

Cray C/C++ Reference Manual (SR–2179)†

Cray C/C++ Ready Reference (SQ–2180)

CrayLibs

Application Programmer's I/O Guide (SG–2168)†

Application Programmer's Library Reference Manual (SR–2165)*

Introducing CrayLibs (IN–2167)†

Math Library Reference Manual (SR–2138)*

Parallel Virtual Machine (PVM) Reference Card (SQ–2512)

PVM and HeNCE Programmer's Manual (SR–2501)†

Scientific Libraries Reference Manual (SR–2081)*

CrayTools

Compiler Information File (CIF) Reference Manual (SR–2401)†

Cray TotalView Debugger Reference Card (SQ–2519)

Guide to Parallel Vector Applications (SG–2182)†

Introducing the Cray TotalView Debugger (IN–2502)†

Introducing the MPP Apprentice Tool (IN–2511)†

Introducing the Program Browser (IN–2140)†

UNICOS CDBX Symbolic Debugger Reference Card (SQ–2110)

UNICOS CDBX Symbolic Debugger Reference Manual (SR–2091)†

UNICOS CDBX Debugger User's Guide (SG–2094)†

General

Cray Research MPP Software Guide (SG–2508)†

CRAY T3D Emulator User's Guide (SG–2500)†

Installing Programming Environment Products (SG–5191)

Migrating to the CRAY T90 Series IEEE Floating Point (SN–2194)†

Optimizing Code on Cray PVP Systems (SG–2192)†

Programming Environment Installation Guide (SG–5217)

Message Passing Toolkit

MPT: PVM Programmer's Manual (SR–2196)†

MPT: MPI Programmer's Manual (SR–2197)†

† Available with CrayDoc
* Man pages available with the `man` command

UNICOS Operating System

General UNICOS

Software Documentation Ready Reference (SQ–2122)

Software Overview for Users (SG–2052)†

User's Guide to Online Information (SG–2143)†

UNICOS `ed` Reference Card (SQ–2055)

UNICOS Environment Variables Ready Reference (SQ–2117)

UNICOS File Formats and Special Files Reference Manual (SR–2014)★

UNICOS Standard Shell Reference Card (SQ–2115)

UNICOS User Commands Ready Reference (SQ–2056)★

UNICOS User Commands Reference Manual (SR–2011)★

UNICOS `vi` Reference Card (SQ–2054)

UNICOS Visual Interfaces User's Guide (SG–3094)†

Networking Feature Use and Administration

Front-end Protocol Internal Reference Manual (SM–0042)

FTA User and Administrator Manual (SG–2144)†

Kerberos Administrator's Guide (SG–2306)†

Kerberos User's Guide (SG–2409)†

Network Queuing System (NQS) User's Guide (SG–2105)†

ONC+ Technology for the UNICOS Operating System (SG–2169)†

Remote Procedure Call (RPC) Reference Manual (SR–2089)†

TCP/IP Network User's Guide (SG–2009)†

UNICOS Networking Facilities Administrator's Guide (SG–2304)†

UNICOS NQS and NQE Administrator's Guide (SG–2305)†

UNICOS Station Call Processor (USCP) Administrator's Guide (SG–2308)†

UNICOS MAX

Cray MPP Loader User's Guide (SG–2514)†

Cray MPP Simulator User's Guide (SG–2503)†

CRAY T3D Administrator's Guide (SG–2507)†★

UNICOS Basic Administration and Operation

CRAY EL Series IOS Commands Ready Reference (SQ–2162)★

CRAY EL Series IOS Commands Reference Manual (SR–2408)★

CRAY EL Series IOS Messages (SQ–2402)

CRAY IOS-V Commands Reference Manual (SR–2170)★

CRAY IOS-V Messages (SQ–2172)†

General UNICOS System Administration (SG–2301)†

`nmake` Reference Card (SQ–2146)

OLNET Online Diagnostic Network Communications Program Maintenance Manual for UNICOS (SD–1021)†

Online Maintenance Tools Guide for Cray PVP Systems (SD–1012)†

UNICOS Basic Administration Guide for CRAY J90 Series and CRAY EL Series (SG–2416)†

UNICOS Configuration Administrator's Guide (SG–2303)†

UNICOS Installation Guide (SG–2112)

UNICOS Installation Guide for the CRAY EL Series (SG–5201)

UNICOS Installation Guide for the CRAY J90 Series (SG–5271)

UNICOS Installation Menu System Reference Card (SQ–2411)

UNICOS Installation/Configuration Menu System User's Guide (SG–2412)†

UNICOS Resource Administration (SG–2302)†

UNICOS Shared File System (SFS) Administrator's Guide (SG–2114)†

UNICOS Feature Use and Administration

UNICOS Administrator Commands Reference Manual (SR–2022)★

UNICOS Administrator Commands Ready Reference (SQ–2413)★

UNICOS Message System Programmer's Guide (SG–2121)†

UNICOS Multilevel Security (MLS) Feature User's Guide (SG–2111)†

UNICOS System Security Overview for Administrators (SG–2141)†

UNICOS Tape Subsystem Administrator's Guide (SG–2307)†

UNICOS Tape Subsystem User's Guide (SG–2051)†

UNICOS under UNICOS Administrator's Guide (SG–2156)†

† Available with CrayDoc
★ Man pages available with the `man` command

UNICOS Operating System (continued)

Programming Support

Segment Loader (SEGLDR) and `ld` Reference Manual (SR–0066)†

SEGLDR Reference Card (SQ–0303)

UNICOS System Libraries Ready Reference (SQ–2147)*

UNICOS System Libraries Reference Manual (SR–2080)*

Programming Support (continued)

UNICOS Macros and Opdefs Reference Manual (SR–2403)*

UNICOS System Calls Reference Manual (SR–2012)*

UNICOS X Window System Reference Manual (SR–2101)†

X Window System Resources Ready Reference (SQ–2123)

Networking Products

ATM

Asynchronous Transfer Mode (ATM) Administrator's Guide (SG–2193)†

fy Driver

fy Driver Administrator's Guide (SG–2132)

HIPPI

HIPPI Channel Administrator's Guide (SG–2159)

Network Queuing Environment (NQE)

NQE User's Guide (SG–2148)†

NQE Ready Reference (SG–2149)

NQE Administration (SG–2150)†

Introducing NQE (IN–2153)†

RQS MVS

RQS User's Guide for IBM MVS Systems (SG–2405)

RQS Administrator's Guide for IBM MVS Systems (SG–2406)

Fiber Distributed Data Interface (FDDI)

Fiber Distributed Data Interface (FDDI) Administrator's Guide (SG–3091)

RQS VMS

RQS/NQC User's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3151)

RQS/NQC Administrator's Guide for DEC OpenVMS/VAX and AXP Systems (SV–3152)

SUPERLINK VM

SUPERLINK VM User's Guide: RQS Component (SI–0170)

SUPERLINK VM Administrator's Guide: RQS Component (SI–0171)

SUPERLINK VMS

SUPERLINK Administrator's Guide for DEC VAX/VMS Systems (SV–3154)

SUPERLINK User's Guide for DEC VAX/VMS Systems (SV–3153)

Operating Systems Supplementary Products

CCMT

Concurrent Maintenance Tools (CCMT) Guide (SD–1027)†

Cray/REELlibrarian (CRL)

Cray/REELlibrarian (CRL) Administrator's Guide (SG–2127)

Cray/REELlibrarian (CRL) User's Guide (SG–2126)

SMARTE

System Maintenance and Remote Testing Environment (SMARTE) Guide (SD–1017)†

Data Migration Facility (DMF)

Cray Data Migration Facility (DMF) Administrator's Guide (SG–2135)†*

Cray Data Migration (DMF) MSP Writer's Guide (SN–2098)†

OWS

Support System Reference Manual (SR–3077)*

Support System Operator's Guide (SG–3078)†

Support System Administrator's Guide (SG–3079)†

Support System Ready Reference (SQ–3080)

† Available with CrayDoc
* Man pages available with the `man` command

New Features

Cray C/C++ Reference Manual SR–2179 2.0

The following features are available with this release of the Cray C/C++ programming environment:

- Exception handling
- New style casts (except `dynamic_cast`)
- Built-in preprocessor
- `volatile` keyword
- Direct compilation (no `.c` file is produced)
- Template instantiation
- Additional support for `restrict`, including marking the `this` pointer as restricted.
- Less restrictive compiler limits
- Improved C++ performance
- Improved error messages
- Binary compatibility with Cray C++ 1.0

Support for CRAY T90 systems with IEEE floating-point hardware is available with this release of Cray Standard C.

Record of Revision

The date of printing or software version number is indicated in the footer.

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.

Preface

This publication documents the commands, directives, language extensions, and other details specific to the Cray Research implementation of the Standard C and C++ languages. 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*, publication SR–2011
- *UNICOS System Libraries Reference Manual*, publication SR–2080
- *UNICOS CDBX Symbolic Debugger Reference Manual*, publication SR–2091
- *UNICOS CDBX Debugger User's Guide*, publication SG–2094
- *Introducing CrayLibs*, publication IN–2167
- *Math Library Reference Manual*, publication SR–2138
- *Scientific Libraries Reference Manual*, publication SR–2081
- *Scientific Libraries User's Guide*, publication SG–2151
- *C++ Language System Release 3.0.1 Library Manual*, publication S3–2131
- *Cray C/C++ Ready Reference*, publication SQ–2180
- *C++ Installation Guide*, publication SG–5161
- *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
- *Application Programmer's Library Reference Manual*, publication SR–2165
- *Application Programmer's I/O Guide*, publication SG–2168
- *Introducing the Program Browser*, publication IN–2140
- *Introducing the Cray TotalView Debugger*, publication IN–2502
- *Compiler Information File (CIF) Reference Manual*, publication SR–2401
- *Guide to Parallel Vector Applications*, publication SG–2182
- *Introducing the MPP Apprentice Tool*, publication IN–2511
- *Optimizing Code on Cray PVP Systems*, publication SG–2192

Ordering publications

The *User Publications Catalog*, publication CP–0099, describes the availability and content of all Cray Research hardware and software documents that are available to customers. Cray Research customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

To order a document, either call the Distribution Center in Mendota Heights, Minnesota, at +1–612–683–5907 or send a facsimile of your request to fax number +1–612–452–0141. Cray Research employees may send electronic mail to `orderdsk` (UNIX system users).

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

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

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names. The following list describes the identifiers: 1 User commands 1B User commands ported from BSD 2 System calls 3 Library routines, macros, and opdefs 4 Devices (special files) 4P Protocols 5 File formats 7 Miscellaneous topics 7D DWB-related information 8 Administrator commands Some internal routines (for example, the `ddcnt`() 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.

Convention	Meaning
[ ]	Brackets enclose optional portions of a command line.
...	Ellipses indicate that a preceding command-line 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, including the following: CRAY C90 series (CRAY C916, CRAY C92A, CRAY C94, CRAY C94A, and CRAY C98 systems) CRAY C90D series (CRAY C92AD, CRAY C94D, CRAY C98D systems) CRAY EL series (CRAY Y-MP EL, CRAY EL92, CRAY EL94, and CRAY EL98 systems) CRAY J90 series (CRAY J916 and CRAY J932 systems) CRAY T90 series (CRAY T94, CRAY T916, and CRAY T932 systems) CRAY Y-MP E series (CRAY Y-MP 2E, CRAY Y-MP 4E, CRAY Y-MP 8E, and CRAY Y-MP 8I systems) CRAY Y-MP M90 series (CRAY Y-MP M92, CRAY Y-MP M94, and CRAY Y-MP M98 systems)
Cray MPP systems	All configurations of Cray massively parallel processing (MPP) systems, including the CRAY T3D series (CRAY T3D MC, CRAY T3D MCA, and CRAY T3D SC systems)

Term	Definition
All Cray Research systems	All configurations of Cray PVP and Cray MPP systems that support this release
SPARC systems	All SPARC platforms that run the Solaris operating system version 2.3 or later

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

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

The UNICOS operating system also supports the optional use of the C shell.

Cray UNICOS Version 9.0 is an X/Open Base 95 branded product.

The POSIX standard uses *utilities* to refer to executable programs that Cray Research documentation usually refers to as *commands*. Both terms may appear in this document.

Online information

The following types of online information products are available to Cray Research customers:

- CrayDoc online documentation reader, which lets you see the text and graphics of a document online. The CrayDoc reader is available on workstations. To start the CrayDoc reader at your workstation, use the `cdoc`(1) command.
- Man pages, which describe a particular element of the UNICOS operating system or a compatible product. To see a detailed description of a particular command or routine, use the `man`(1) command.
- UNICOS message system, which provides explanations of error messages. To see an explanation of a message, use the `explain`(1) command.

- Cray Research online glossary, which explains the terms used in a document. To get a definition, use the `define`(1) command.
- `xhelp` help facility. This online help system is available within tools such as the Program Browser (`xbrowse`) and the MPP Apprentice tool.

For detailed information on these topics, see the *User's Guide to Online Information*, publication SG–2143.

Reader comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. You can contact us in any of the following ways:

- Send us electronic mail from a UNICOS or UNIX system, using the following UUCP address:

  `uunet!cray!publications`

- Send us electronic mail from any system connected to the Internet, using the following Internet address:

  `publications@timbuk.cray.com`

- Contact your Cray Research representative and ask that a Software Problem Report (SPR) be filed. Use `PUBLICATIONS` for the group name, `PUBS` for the command, and `NO-LICENSE` for the release name.
- Call our Software Publications Group in Eagan, Minnesota, through the Customer Service Call Center, using either of the following numbers:

  1–800–950–2729 (toll free from the United States and Canada)

  +1–612–683–5600

- Send a facsimile of your comments to the attention of "Software Publications Group" in Eagan, Minnesota, at fax number +1–612–683–5599.

- Use the postage-paid Reader's Comment Form at the back of the printed document.

We value your comments and will respond to them promptly.

Contents

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.

> **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.

Manual contents

1.1

This manual contains the following sections:

Section	Description
1	Describes the compilers and the differences between them.
2	Describes UNICOS information, compiler commands, and files.
3	Describes compiler behavior that the standard requires to be defined for each particular implementation and discusses environmental limits imposed on programs by the implementation.
4	Describes the use of `#pragma` directives.
5	Describes the concept of templates and their use.
6	Describes predefined macros.

Section	Description
7	Describes intrinsic functions by platform.
8	Describes the tools and processes used in debugging C/C++ code.
9	Describes calls to assembly or Fortran functions from Cray C/C++ and calls to Cray C/C++ functions from Fortran and assembly language.
10	Describes Cray Research extensions to the C and C++ standards.
A	Lists the libraries and loaders that are available to Cray C/C++.
B	Describes the features of the C++ language that are accepted by the Cray C++ compiler, including C++ anachronisms.
C	Describes how to control and use messages issued by the compiler using the UNICOS message system.

General compiler description

1.2

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.

Cray C++ compiler

1.2.1

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 subsection 2.1, page 6, and on the `CC`(1) man page. Command-line examples are shown in subsection 2.4, page 55.

Cray Standard C compiler

1.2.2

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`(1) or `c89`(1) in this manual, but optionally renamed at individual sites. The `cc`(1) and `c89`(1) commands are described in subsection 2.1, page 6, and on the `CC`(1) man page. Command-line examples are shown in subsection 2.4, page 55.

Compiler Commands [2]

This section describes the commands and environment variables necessary to execute the compilers associated with the Cray C++ programming environment. The commands are as follows:

- `CC`(1)

  The `CC` command invokes the Cray C++ compiler.

- `cc`(1)

  The `cc` command invokes the Cray Standard C compiler.

- `c89`(1)

  The `c89` command invokes the Cray Standard C compiler.

- `cpp`(1)

  The `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, `ld`(1) or `mppld`(1), which creates an executable file. (The library and loader interface is described in subsection A.2, page 203.) 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
```

CC command

2.1

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*] [`-o` *outfile*] [`-O` *level*] [`-P`] [`-s`] [`-S`] [`-U` *macro*]
[`-V`] [`-W`*phase* `,`[*"opt... "*]] [`-X`*npes*] [`-Y`*phase* `,`*dirname*] [`-#`] [`-##`] [`-###`] *files ...*

Options from the preceding format are described in subsection 2.3.

cc command

2.2

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 ] [-o outfile ] [-O level ] [-P] [-s] [-S]
   [-U macro ] [-V] [-Wphase ,["opt... "]] [-Xnpes ] [-Yphase ,dirname ] [-#] [-##]
   [-###] files ...
```

Section 2

Options from the preceding format are described in subsection 2.3, page 8.

c89 command

2.2.1

The `c89` command invokes the Cray Standard C compiler. This command is a subset of the `cc`(1) 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 ...
```

Options from the preceding format are described in subsection 2.3, page 8.

Section 2

cpp command

2.2.2

The `cpp` command explicitly invokes the preprocessor component of the Cray Standard C compiler. All `cpp` options are available from the `cc` and `c89` commands with the exception of `-M` and `-N`. The `-M` and `-N` options can also be invoked from the `cc` command line by means of the `-Wp` option as shown in the following example:

```
cc -Wp,-M,-N myfile.c
```

The `cpp` command format is as follows:

> `cpp` [`-C`] [`-D` *macro* [=*def*]] [`-E`] [`-I` *incldir*] [`-L` *libdir*] [`-M`] [`-N`] [`-P`]
> [`-U` *macro*] [`-V`] [`-Y`*phase* `,`*dirname*] [`-#`] [`-##`] [`-###`] *files* ...

Options from the preceding format are described in subsection 2.3, page 8.

CC, cc, c89, and cpp command options

2.3

This subsection describes options for the `CC`, `cc`, `c89`, and `cpp` commands. Options other than those described here are passed to the loader; see the `ld`(1) and `mppld`(1) man pages for loader options.

At the beginning of each subsection, information is included about exceptions to the default option, the systems that use the option (PVP and/or MPP), the type of floating-point representation used (IEEE and/or Cray), and the commands that will accept the option (any or all of `CC`, `cc`, `c89`, or `cpp`). Unless otherwise noted, the following default information applies to each option:

Default option:	None
Systems:	PVP, MPP
Floating-point:	IEEE, Cray
Command:	`CC`, `cc`, `c89`

`-c`: object file

2.3.1

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.

`-C`: retain comments

2.3.2

Command: `CC`, `cc`, `cpp`

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`, `cc`, and `c89` commands.

`-d` *string*: send string to loader

2.3.3

Command: `CC`, `cc`

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. On PVP systems the loader is SEGLDR (see the `ld`(1) man page); on MPP systems it is MPPLDR (see the `mppld`(1) man page).

`-D` *macro*[=*def*]: define macro

2.3.4

Command: `CC`, `cc`, `c89`, `cpp`

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`.

Section 2

Predefined macros also exist; these are described in section 6, page 129. Any predefined macro except those required by the standard (see subsection 6.1, page 129) can be redefined by the −D option. If the same macro is specified, −U overrides −D, regardless of the order in which they are specified on the command line.

−E: preprocessor only

2.3.5

Command: CC, cc, c89, cpp

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 −P, −S, or −c options.

If the −E option is specified on the cpp command line, it inserts the appropriate #line directives in the preprocessed output. The −E option takes precedence over the −P option.

−F: Flowtrace processing

2.3.6

Systems: PVP only
Command: CC, cc

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.

`-G` *level* and `-g`: enable symbolic debugging

2.3.7

Command: `CC`, `cc` (`-G`)
`CC`, `cc`, `c89` (`-g`)

The `-g` and `-G` *level* options enable the generation of debugging information that is used by symbolic debuggers such as `cdbx`(1) and `totalview`(1). These options allow debugging with breakpoints; *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 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.

Unlike `-Gn`, `-g` disables all optimizations and implies `-O0`.

Debugging is described in more detail in section 8, page 147.

`-h` *arg*: enable compiler actions

2.3.8

The `-h` *arg* option enables specific compiler actions. Arguments to the `-h` option are described in subsection 2.3.27, page 22.

`-I` *incldir*: directory for `#include` files

2.3.9

Default option: See the following description
Command: `CC`, `cc`, `c89`, `cpp`

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. Default directories.

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

1. Directories named in `-I` options, in command-line order.
2. Default directories.

Default directories are searched in the following order:

1. Site- and release-specific C++ include files directories (C++ only).
2. Directory `/usr/include/CC` (C++ only).
3. Directory `/usr/include`.

If the `-I` option specifies a directory name that does not begin with a slash (/), 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- and release-specific C++ include files directories (C++ only).
5. `/usr/include/CC` (C++ only).
6. `/usr/include`.

`-l` *libfile*: library files

2.3.10

Default option: See the following description

The `-l` *libfile* option identifies library files to be loaded. If *libfile* begins with a period (`.`) or forward 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 as follows:

Cray MPP systems	Cray PVP systems
`libC.a` (C++ only)	`libC.a` (C++ only)
`libu.a`	`libu.a`
`libm.a`	`libm.a`
`libc.a`	`libc.a`
`libpvm3.a`	`libf.a`
`libsma.a`	`libfi.a`
`libf.a`	`libsci.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.

Section 2

`-L` *libdir*: library file directory

2.3.11

Default option: See the following description

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 forward 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- and release-specific libraries directories (C++ only)
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. The library and loader interface is described in appendix A, page 203.

-M: include and inline file dependencies
2.3.12

Command: `cpp` only

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`.

-N: enable old style (K & R) preprocessing
2.3.13

Command: `cpp` only

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.

-o *outfile*: specify output file name
2.3.14

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.

-O *level*: optimization level
2.3.15

Default option: As shown for `-h` options in the following table

The `-O` *level* option specifies the optimization level for a group of compiler features. 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
```

Section 2

Optimization features specified by `-O` are as follows (`task` and `vector` are ignored on Cray MPP systems):

`-h` option	Description location
`-h inline`*n*	Subsection 2.3.27.26, page 31
`-h scalar`*n*	Subsection 2.3.27.44, page 42
`-h task`*n*	Subsection 2.3.27.50, page 45
`-h vector`*n*	Subsection 2.3.27.57, page 48

`-P`: preprocessor only

2.3.16

Command: `CC`, `cc`, `cpp`

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 `-S` or `-c`.

When the `-P` option is specified on the `cpp` command line, it is a `no-op`.

`-s`: remove symbolic information

2.3.17

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.

−S: assembly language output

2.3.18

Command: CC, cc

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. This option takes precedence over −c.

−T[*cpu*],[*hdw*]: CPU targeting

2.3.19

The −T option has been deprecated and will not be available with the next major release of the compiler. See subsection 2.3.30, page 50, for more information.

−U *macro*: undefine macro

2.3.20

Command: CC, cc, c89, cpp

The −U option removes any initial definition of *macro* . Any predefined macro except those required by the standard (see subsection 6.1, page 129) 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 section 6, page 129. Macros defined in the standard headers are not predefined macros and are not affected by the −U option.

Section 2

–V: display compiler version

2.3.21

Command: CC, cc, cpp

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, an internal version number in parentheses (cc only), and the current date and time, as shown in the following example:

```
% CC -V
Cray C++ Version 2.0.0  09/30/95  09:31:12
```

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, as shown in the following example:

```
% cc -V hello.c
Cray Standard C Version 4.0.0 (534083) Sep 30 1995  10:24:48

     Compiler Generation Date   = Sep 30 1995
     Compilation Execution Time = 0.069350 seconds
     Maximum Field Length       = 1201462 decimal words
     Code:  14 decimal words    Data:  10 decimal words
```

–W*phase* **[“*opt...*”]: pass options to system phases**

2.3.22

Command: `CC`, `cc`

The `–W`*phase* option passes arguments directly to a phase of the compiling system. For example, this option allows access to the preprocessor `–M` and `–N` options, described on page 8, through the `CC` or `cc` command lines. The `–W` option appears with argument *phase* to indicate system phases as follows:

Option	System phase	Command
`–Wp`	Preprocessor	`cpp`(1)
`–W0`	Compiler	
`–Wa`	Assembler	`as`(1) on Cray PVP, `cam`(1) on MPP
`–Wl`	Loader	`ld`(1) on PVP, `mppld`(1) on MPP

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 backslash escape 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.

Section 2

Section 2

–X*npes*: number of processors

2.3.23

Systems: MPP only
Command: CC, cc

The -X option specifies how many processing elements (PEs) are to be used on a Cray MPP system. The *npes* argument specifies the number of PEs and has no default value, it must be explicitly set.

The option is passed from the command to both the compiler and the loader. If the option is seen by the compiler, it becomes a compile-time value and cannot be changed at load time. If the option is seen by the loader, 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 seen first 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. For example:

```
cc -X16 file1.c                    /* OK */
cc -W0,-X8 -Wl,-X4 file2.c         /* not OK */
cc -X2 -c file3.c; cc -X1 file3.o /* not OK */
```

Note: If the `-c` option is specified, the loader is not called.

2.3.24 `-Y`*phase*, *dirname*: directory for phases of processing

Command: `CC`, `cc`, `c89`, `cpp`

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	`cpp`(1)
`0`	Compiler	
`a`	Assembler	`as`(1) on Cray PVP, `cam`(1) on Cray MPP
`l`	Loader	`ld`(1) on Cray PVP, `mppld`(1) on Cray MPP

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

2.3.25 `-#`: show compilation phase

Command: `CC`, `cc`, `cpp`

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

−## and −###: display during compilation
2.3.26

Command: CC, cc, cpp

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.

−h option arguments
2.3.27

The arguments to the −h option, described in the following subsections, can be used to direct specific compiler actions. Multiple arguments must be separated by a comma and have no intervening spaces.

−h [no]abort: abort on error
2.3.27.1

Default option: −h noabort
Command: CC, cc

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

−h [no]aggress: aggressive optimization
2.3.27.2

Default option: −h noaggress
Command: CC, cc

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 Cray PVP systems this also disables the limit on the number of vector updates in a single loop. On Cray MPP systems, aggressive optimization is used to assign registers and schedule instructions. This option is affected by the `-h vector`*n* option (see subsection 2.3.27.57, page 48).

2.3.27.3 `-h [no]align`: function alignment

Default option: `-h noalign`
Systems: PVP only
Command: `CC`, `cc`

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.

2.3.27.4 `-h [no]anachronisms`: accept certain C++ anachronisms

Default option: `-h anachronisms`
Command: `CC`

The `-h [no]anachronisms` option enables or disables anachronisms in Cray C++. These anachronisms are described in subsection B.2, page 210, of this manual. This option is overridden by `-h conform`.

2.3.27.5 `-h [no]apprentice`: enable MPP Apprentice tool

Default option: `-h noapprentice`
Systems: MPP only
Command: `CC`, `cc`

The `-h apprentice` option enables or disables the creation of a compiler information file (CIF) for each source file compiled. The CIF is used by the MPP Apprentice tool.

The MPP Apprentice tool helps you tune the performance of MPP applications. It is documented in *Introducing the MPP Apprentice Tool*, publication IN–2511.

2.3.27.6 `-h [no]atexpert`: generate code for the ATExpert tool

Default option: `-h noatexpert`
Systems: PVP only
Command: `CC`, `cc`

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`(1) man page.

2.3.27.7 `-h [no]autoinstantiate`: enable/disable automatic instantiation

Default option: `-h noautoinstantiate`
Command: `CC`

The `-h [no]autoinstantiate` option enables or disables automatic instantiation of templates by the Cray C++ compiler. Automatic instantiation is described in detail in subsection 5.1, page 122.

2.3.27.8 `-h [no]bl`: aggressive bottom loading

Default option: `-h nobl`
Command: `CC`, `cc`

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 subsection 2.3.27.44, page 42).

`-h [no]bounds`: bounds checking
2.3.27.9

Default option: `-h nobounds`
Command: `cc`

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.

`-h [no]calchars`: permit restricted characters
2.3.27.10

Default option: `-h nocalchars`
Command: `CC`, `cc`

Caution: Use this option only 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.

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.

`-h cfront`: accept constructs previously accepted by `cfront`
2.3.27.11

Command: `CC`

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. For more information, see subsection B.4, page 215.

2.3.27.12 `-h char`: treat plain `char` as a distinct type

Command: `CC`

The `-h char` option tells the C++ compiler to treat plain `char` as a distinct type and allow the use of signed, unsigned and plain `char` types within a program. This eliminates duplicate definitions at linkage or errors at compilation.

2.3.27.13 `-h cif=`*options*: compiler information file

Default option: no CIF
Command: `cc`

The `-h cif=`*options* option creates a compiler information file (CIF), which is directed to `stdout` unless the `f` option is used (described in the following list). The CIF includes the names of all variables, their types, line numbers, locations, and so forth. The CIF is used by Cray Research tools such as `xbrowse`(1) and `apprentice`(1), but these tools do not require the use of the `-h cif` option. For more information about the CIF, see the *Compiler Information File (CIF) Reference Manual*, publication SR–2401. The *options* argument can be any combination of the following:

options	Description
`1`	Limits analysis to the parsing phase of the compiler.
`2`	Expands analysis to all phases of the compiler. A `.o` file is not produced.
`3`	Same as level 2, but a `.o` file is produced.
`f`	Directs the CIF output to a file. The file name corresponds to the name of the source file with the `.T` suffix substituted for `.c`.

Note: When `-h cif=1` or `-h cif=2` is used, no object file (`.o` suffix) is created unless the `-c` option is also specified.

Section 2

2.3.27.14 `-h common`: overriding task common

Systems: PVP only
Command: `CC`, `cc`

The `-h common` option gives `common` (as opposed to `taskcommon`) status to all global objects in the program. For information about `taskcommon` status, see subsection 2.3.27.51, page 46.

2.3.27.15 `-h [no]conform`: specify conformance to standard

Default option: `-h noconform`
Command: `CC`, `cc`

The `-h conform` option specifies strict conformance to the ISO C standard or the draft ISO C++ standard. The `-h noconform` option specifies partial conformance to the standard. `-h exceptions` is enabled by `-h conform`.

> **Note:** The `c89` command does not accept the `-h conform` option when issued on the command line. It is enabled by default when the command is issued.

2.3.27.16 `-h errorlimit[=n]`: error message limit

Default option: `-h errorlimit=70` (C)
`-h errorlimit=100` (C++)
Command: `CC`, `cc`

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

Section 2

2.3.27.17 `-h [no]exceptions`: enable or disable support for exception handling

Default option:	`-h noexceptions`
Command:	`CC`

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`.

2.3.27.18 `-h [no]fastfpdivide`: change floating-point divide to multiply-by-reciprocal

Default option:	`-h fastfpdivide`
Floating-point:	IEEE only
Command:	`CC`, `cc`

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, specified by `-O` (see subsection 2.3.15, page 15) or `-h scalar`*n* (see subsection 2.3.27.44, page 42).

2.3.27.19 `-h [no]fastmd`: shorter code for `int` variables

Default option: `-h fastmd`
Systems: PVP only
Command: `CC`, `cc`

The `-h fastmd` option generates shorter code sequences for `int` variables when doing multiply, divide, or comparison operations, but allows for only 46 bits of significance. With `-h nofastmd`, this is disabled. This option is affected by the scalar optimization level, specified by `-O` (see subsection 2.3.15, page 15) or `-h scalar`*n* (see subsection 2.3.27.44, page 42).

2.3.27.20 `-h [no]fastmodulus`: faster modulus code

Default option: `-h nofastmodulus`
Systems: PVP only
Command: `CC`, `cc`

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, specified by `-O` (see subsection 2.3.15, page 15) or `-h scalar`*n* (see subsection 2.3.27.44, page 42).

2.3.27.21 `-h feonly`: limits the C++ compiler to syntax checking only

Command: `CC`

The `-h feonly` option limits the C++ compiler to syntax checking. The optimizer and code generator are not executed.

`-h forcevtbl`: forces definition of virtual function tables 2.3.27.22

Command: `CC`

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

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).

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

See also the `-h suppressvtbl` option in subsection 2.3.27.49, page 44, for more information.

`-h ident=`*name*: specify ident name 2.3.27.23

Default option: File name specified on the command line
Command: `CC`, `cc`

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.

2.3.27.24 `-h [no]implicitinclude`: enable/disable implicit inclusion

Default option: `-h implicitinclude`
Command: `CC`

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. Implicit inclusion is described in subsection 5.4, page 128.

2.3.27.25 `-h [no]indef`: initialize stack to undefined values

Default option: `-h noindef`
Command: `CC`, `cc`

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. This option is especially useful for debugging tasked codes.

2.3.27.26 `-h inline`*n*: automatic inlining level

Default option: `-h inline1`
Command: `CC`, `cc`

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	Automatic inlining
`0`	No inlining is performed.
`1`	Conservative inlining; performed only on functions implicitly marked by the `inline` keyword or a `#pragma _CRI inline` directive or on functions defined inside a class definition.
`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 `noinline` directive.

2.3.27.27 `-h inlinefrom=`*file*: select file to be inlined

Command: `CC`

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.

2.3.27.28 `-h instantiate=`*mode*: set instantiation mode

Default option: `-h instantiate=none`
Command: `CC`

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`. Template instantiation is described in detail in section 5, page 121. Detailed information regarding instantiation modes and this option are described in subsection 5.2, page 124.

2.3.27.29 `-h` [`no`]`intrinsics`: use intrinisic functions

Default option: `-h intrinsics`
Command: `CC`, `cc`

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 section 7, page 133.

`-h [no]ivdep`: vector dependency analysis

2.3.27.30

Default option: `-h noivdep`
Systems: PVP only
Command: `CC`, `cc`

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.

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 `ivdep` directive, as discussed in subsection 4.5.1, page 86.

`-h [no]jump`: use jumps instead of branches

2.3.27.31

Default option: `-h jump`
Systems: MPP only
Command: `CC`, `cc`

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. A loader message similar to the following indicates the need for `-h jump`:

```
mppldr-331 _mppldr: CAUTION
DEX expression 7 module 'xxx$c' calculated a relative
branch target too distant.
The last symbol referenced was 'abc'.
```

2.3.27.32 -h keep=*file*: retain object files

Command: CC

When the -h keep=*file* option is specified, the static constructor/destructor .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 program 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 ld(1) command.

2.3.27.33 -h [no]listing: listing file

Default option: -h nolisting
Systems: PVP only
Command: CC, cc

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 .s suffix replaced by a .L.

`-h matherror=`*method*: math function error processing
2.3.27.34

Default option: `-h matherror=abort`
Command: `CC`, `cc`

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 `-h conform`.

`-h [no]message=`*n*[:*n*...]: error message printing
2.3.27.35

Default option: Determined by `-h msglevel_`*n*
Command: `CC`, `cc`

The `-h 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-172` and `cc-65`, specify the following:

```
-h nomessage=172:65
```

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.

2.3.27.36 `-h msglevel_`*n*: severity level of messages

Default option: `-h msglevel_3`
Command: `CC`, `cc`

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).

2.3.27.37 `-h [no]overindex`: array subscripts that are outside declared bounds

Default option: `-h nooverindex`
Command: `CC`, `cc` (Not available in Cray Standard C 4.0)

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.

2.3.27.38 `-h [no]pattern`: vectorize with pattern matching

Default option: `-h pattern`
Command: `CC`, `cc`

The `-h [no]pattern` option globally enables or disables pattern matching. Pattern matching is on by default, but only takes effect when `-h vector3` or `-h task3` are specified.

`-h [no]pragma=`*name*[:*name*...]: directive processing

2.3.27.39

Default option: `-h pragma`
Command: `CC, cc`

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 1 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 1. `-h pragma` directive processing

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

† The `noreduction` directive is a scalar directive in Cray Standard C 5.0 a Cray C++ 2.0. It is a vectorization directive in Cray Standard C 4.0.

You must use care when using this option to enable or disable individual directives. 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.

2.3.27.40 `-h [no]reduction`: vectorizing reduction loops

Default option:	`-h reduction`
Systems:	PVP only
Command:	`CC`, `cc`

The `-h reduction` option instructs the compiler to vectorize all reduction loops. With `-h noreduction`, reduction loops are not vectorized. This option is affected by the `-h vector`*n* option (see subsection 2.3.27.57, page 48). Reduction loops and the `noreduction` directive are discussed in subsection 4.7.3, page 110.

2.3.27.41 `-h report=`*args*: report messages

Default option:	`-h report`
Systems:	PVP only
Command:	`CC, cc`

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`	Selects inlining optimization messages
`s`	Selects scalar optimization messages
`t`	Selects tasking optimization messages

args	Description
`v`	Selects 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 `stdout`.

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 `stdout`:

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

2.3.27.42 `-h restrict=`*args*: restricted pointers

Command: `CC`, `cc`

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 PVP systems.

Caution: Like `-h ivdep`, the arguments make assertions about your program that, if incorrect, can introduce undefined behavior. The `-h restrict=f` and `-h restrict=t` options are subject to the analogous restriction, with “pointer function parameter” replacing “pointer”. 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

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 `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.

2.3.27.43 `-h [no]rounddiv`: strong rounding

Default option: `-h norounddiv`

Floating-point: Cray only

Command: `CC, cc`

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, in turn, ensures that, if these floating-point values are converted back into integer values, they will

Section 2

represent the expected results. For example, results such as 2.99999..., when converted, will be treated as 3.

2.3.27.44 `-h scalarn`: scalar optimization level

Default option: `-h scalar1`
Command: `CC`, `cc`

The `-h scalarn` option specifies the level of automatic scalar optimization to be performed. Scalar optimization directives are unaffected by this option (see subsection 4.7, page 105).

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. If the `cc`(1) command is used, tail recursion is enabled. If the `CC`(1) command is specified, tail recursion is an inlining optimization.
`2`	Moderate automatic scalar optimization. The scalar optimizations specified by `scalar1` are performed, and aggressive bottom loading (`-h bl`) is added.
`3`	Aggressive automatic scalar optimization. The scalar optimizations specified by `scalar2` are performed and `-h fastmodulus` is implied.

2.3.27.45 `-h [no]signedshifts`: affect the result of the right shift operator

Default option:	`-h signedshifts`
Systems:	MPP only
Floating-point:	IEEE only
Command:	`CC`, `cc`

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 Cray MPP systems as on Cray PVP systems.

2.3.27.46 `-h [no]split`: loop splitting for CRAY T3E stream buffers

Default option:	`-h nosplit`
Systems:	CRAY T3E only
Command:	`CC`, `cc`

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 CRAY T3E systems by reducing thrashing of CRAY T3E 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 subsection 4.7.4, page 111.

Section 2

2.3.27.47 `-h [no]stack`: memory allocation

Default option:	`-h stack`
Systems:	PVP only
Command:	`CC`, `cc`

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` option.

2.3.27.48 `-h [no]stdc`: conformance to the standard

Default option:	`-h nostdc`
Command:	`cc`

The `-h stdc` option specifies strict conformance to the ISO C standard. `-h nostdc` specifies partial conformance to the standard.

The `-h [no]conform` option is preferred over `-h [no]stdc` because it is available in both Cray Standard C and Cray C++. The `-h [no]conform` option is described in subsection 2.3.27.15, page 27.

> **Note:** The `c89`(1) command does not accept the `-h stdc` option when issued on the command line. It is enabled by default when the command is issued.

2.3.27.49 `-h suppressvtbl`: suppress definition of virtual function tables

Command:	`CC`

The `-h suppressvtbl` option suppresses 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 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.

See also the `-h forcevtbl` option in subsection 2.3.27.22, page 30.

2.3.27.50 `-h task`*n*: tasking level

Default option: `-h task2`
Command: `CC`, `cc`

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 subsection 4.6, page 90.

Note: The `-h task`*n* option is accepted and ignored on MPP 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.

Section 2

`-h taskcommon`: task common

2.3.27.51

Default option: `-h taskcommon`
Systems: PVP only
Command: `CC`, `cc`

The `-h taskcommon` option gives task common status to all global objects in the program. It has no effect if `-h nostack` is specified. Tasking and tasking directives are described in subsection 4.6, page 90.

`-h [no]taskinner`

2.3.27.52

Default option: `-h notaskinner`
Systems: PVP only
Command: `CC`, `cc`

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. `-h taskinner` 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 (see subsection 2.3.27.50, page 45) must also be specified for this option to take effect.

`-h [no]threshold`

2.3.27.53

Default option: `-h threshold`
Systems: PVP only
Command: `CC`, `cc`

The `-h [no]threshold` option enables or disables generation of run-time threshold testing for autotasked loops. Aggressive Autotasking (see subsection 2.3.27.50, page 45) must also be specified for this option to take effect.

`-h [no]tolerant`: tolerate nonstandard constructs
2.3.27.54

Default option: `-h notolerant`
Command: `cc`

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 which inhibits optimization.

`-h [no]trunc[=`*n*`]`: truncation
2.3.27.55

Default option: `-h notrunc`
Floating-point: Cray only
Command: `CC`, `cc`

The `-h trunc[=`*n*`]` option specifies truncation of the last n bits (range of n: $0 \le n \le 47$) 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. `-h notrunc` generates assembler rounding instructions for floating arithmetic.

Section 2

`-h [no]unroll`: loop unrolling
2.3.27.56

Default option:	`-h nounroll`
Systems:	MPP only
Command:	`CC`, `cc`

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 on 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 subsection 4.7.7, page 115.

> **Note:** On Cray PVP systems, `-h unroll` is enabled at all times.

`-h vector`*n*: vectorization level
2.3.27.57

Default option:	`-h vector2`
Systems:	PVP only
Command:	`CC`, `cc`

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.

Argument *n* can be one of the following:

n	Description
`0`	No automatic vectorization.
`1`	Conservative automatic vectorization. Automatic vectorization is performed. Search loops and reduction loops are not vectorized.

n	Description
`2`	Moderate automatic vectorization. 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 the `-h aggress` option is enabled. In addition, 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).

Vectorization directives are described in subsection 4.5, page 86.

Note: The `-h vector`*n* option is accepted and ignored on MPP systems.

2.3.27.58 `-h [no]vsearch`: vectorize search loops

Default option: `-h vsearch`

Systems: PVP only

Command: `CC`, `cc`

The `-h vsearch` option enables vectorization of all search loops. With `-h novsearch`, the default vectorization level applies. The `novsearch` directive is discussed in subsection 4.5.3, page 88. This option is affected by the `-h vector`*n* option (see subsection 2.3.27.57, page 48).

2.3.27.59 `-h [no]zeroinc`: increment loops by 0

Default option: `-h nozeroinc`

Command: `CC`, `cc`

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 vectorized and tasked 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 subsection 2.3.27.44, page 42).

2.3.28 Conflicting options

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.

The following examples illustrate the use of conflicting options:

- In this example, `-h nofastmd` overrides `-h fastmd`.

```
CC -h fastmd,nofastmd myfile.c
```

- In this example, `-h vector2` overrides the earlier vector optimization level 3 implied by the `-O3` option.

```
CC -O3 -h vector2 myfile.c
```

2.3.29 Conflicts between options and `#pragma` directives

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.

2.3.30 Deprecated options

The options listed in the following subsections are deprecated and will not be available in the next major release of the compiler.

2.3.30.1 `-T`[*cpu*][`,` *hdw*]: CPU targeting

Default:	Host machine, unless the `TARGET` environment variable is set
Command:	`CC`, `cc`, `c89`, `cpp`

The `-T` option designates the CPU type and hardware characteristics for which your program is to be compiled. Enter the `target`(1) command with no arguments to see your system's configuration. See the `target`(1) man page for more information.

It is recommended that these settings be made using the `TARGET` environment variable rather than using the `-T` option, and that this variable be used only within a script or `make` file, so that it does not affect other programs in the same session.

The *cpu* argument should be used only for cross-compiling. It can be `cray-t3d`, `cray-t3e`, `cray-ymp`, `cray-c90`, `cray-j90`, or `cray-ts`. For cross-compiling, the compiler assumes the default hardware characteristics for the specified CPU type, so you must specify all nondefault hardware characteristics that you require. To see the defaults for a given *cpu* type, enter `target` *cpu*.

After you have compiled your program for another machine, it is suggested that you load the compiled program on the target machine, to ensure that the correct libraries are used. If you load the program on the host machine, you must determine which libraries will be needed on the target machine and how they should be specified; these requirements vary by site and by software release.

To change a hardware characteristic on your system to compile a program to run on the same system (normally to disable a specific hardware characteristic), do not use the *cpu* argument, but include a comma before the *hdw* arguments, in the form `-T,`*hdw* `,`*hdw*. *hdw* arguments are as follows:

All Cray Research systems:

hdw	Target machine does/does not have:
memsize=*wds*	Memory size in words. K following *wds* indicates units of 1024; M indicates units of 1,048,576.

Cray PVP systems only:

hdw	Target machine does/does not have:
memspeed=*cp*	Memory speed in clock periods
[no]avl	Additional vector logical hardware
[no]bdm	Bidirectional memory hardware
[no]cigs	Gather -scatter/compressed index hardware
[no]readvl	Read vector length hardware
[no]vpop	Vector population count functional unit
[no]vrecur	Vector recursion hardware
[no]ieee	IEEE floating-point hardware (CRAY T90 only)

2.3.30.2 -h [no]fctnproto:

Default option: -h fctnproto

Command: CC

This option controls whether or not all functions and function calls will be turned into the old-style unprototyped form. Use of this option is not necessary unless you want to link with code that was compiled by using the Cray C++ 1.0 +a1 option.

2.3.30.3 `-h port=`*args*: messages for porting

Command: `cc`

The `-h port=`*args* option selects porting messages specified in *args*, and lets you direct the specified messages to a file. In the following explanations, PCC refers to the earlier compiler known as the Cray Portable C compiler; SCC refers to the Cray Standard C compiler described in this manual. *args* can be any combination of the following:

args	Description
`i`	Indicates code constructs for which the meaning or behavior is, according to the standard, either implementation defined or undefined.
`n`	Indicates features that are required by the standard and are in SCC but not in PCC.
`s`	Indicates code constructs that are valid for both SCC and PCC but that are interpreted differently (silent changes).
`t`	(Cray PVP systems only) Prints a message for each declaration of a nonstack variable. This lets you identify, in a tasked program, any variables that might require an explicit directive to place them properly in common or task common storage.
`u`	Indicates all variables that are declared but not referenced.
`x`	Indicates features that are Cray Research extensions to the standard and are in SCC.
`f`	Writes porting messages to file *file*`.V`, where *file* is the *file*`.c` specified on the command line. Messages are written to `stdout` if the `f` option is not specified.

No spaces are allowed around the equal sign (=) or any of the *args*. No object or executable code is generated when this option is specified.

Section 2

2.3.30.4 Cray C++ 1.0 deprecated options

The following options, available with Cray C++ 1.0, have been deprecated and will not be available with the next major release of the compiler:

Option	Recommended replacement
`-flow`	`-F` (see subsection 2.3.6, page 10).
`-ispace`	`-h inline`*n* (see subsection 2.3.27.26, page 31).
`-ispeed`	`-h inline`*n* (see subsection 2.3.27.26, page 31).
`-keep`	`-h keep=` (see subsection 2.3.27.32, page 35).
`-p`	See section 5, page 121 for information on template instantiation.
`-$`	`-h calchars` (see subsection 2.3.27.10, page 25).
`-.suffix`	None.
`+a`	None.
`+d`	`-h inline0` (see subsection 2.3.27.26, page 31).
`+e0`	`-h suppressvtbl` (see subsection 2.3.27.49, page 44).
`+e1`	`-h forcevtbl` (see subsection 2.3.27.22, page 30).
`+I`	`-h inline3` (see subsection 2.3.27.26, page 31).
`+i`	None.
`+p`	`-h [no]anachronisms` (see subsection 2.3.27.4, page 23).
`+s`	See section 5, page 121, for information on template instantiation.
`+t`	See section 5, page 121, for information on template instantiation.

Option	Recommended replacement
`+T`	See section 5, page 121, for information on template instantiation.
`+w`	`-h msglevel_`*n* (see subsection 2.3.27.36, page 37).
`+x`	None.

2.4 Command-line examples

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

- The following example compiles `myprog.C` on Cray MPP 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 a loader directive (`-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 `stdout`.

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

2.5 Environment variables

The environment variables listed below are used during compilation.

Variable	Description
`CRI_CC_OPTIONS`	(Cray C++ only) 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. See appendix C, page 221.
`MSG_FORMAT`	Controls the format in which you receive compiler messages. See appendix C, page 221.
`NLSPATH`	Specifies the message system catalogs that should be used. See appendix C, page 221.
`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`(1) and `sh`(1) man pages for more information.

Implementation-defined Behavior and Compiler Limits [3]

This section describes compiler behavior that is defined by the implementation according to the C and/or C++ standards. The C standard requires that the behavior of each particular implementation be documented. This section also discusses environmental limits that are imposed on programs by the implementation.

Implementation-defined behavior

3.1

The C standard defines 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 C and Cray C++ compilers for these cases is summarized in this section.

Section 3

Messages

3.1.1

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 221.

Environment

3.1.2

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` refer to interactive devices and buffer them accordingly . For further information, see the description of I/O in the *UNICOS Systems Libraries Reference Manual*, publication SR–2080.

Identifiers

3.1.2.1

The *identifier* (as defined by the standards) is merely a sequence of letters and digits. Specific uses of identifiers are called *names* .

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.

Section 3

Types

3.1.2.2

Table 2 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 2. Thus, the `sizeof` operator will return a size that is equal to the value in the Memory column of Table 2, divided by 8 (the number of bits in a byte).

Table 2. Cray Research systems data type mapping

Type	Cray PVP systems Representation (in bits)	Cray PVP systems Memory (in bits)	Cray MPP systems Representation (in bits)	Cray MPP systems Memory (in bits)
`char`	8	8	8	8
`short`	32 CRAY T90: 46/64 †	64	32	32
`int`	46/64 †	64	64	64
`long`	64	64	64	64
`float`	64	64	32	32
`double`	64	64	64	64
`long double`	128	128	64	64
`float complex`††	128 (64 each part)	128	64 (32 each part)	64
`double complex`††	128 (64 each part)	128	128 (64 each part)	128
`void` and `char` pointers and all pointers on CRAY T90 systems	64	64	64	64
Other pointers	32 CRAY T90: 64	64	64	64

† Depends on use of the `-h [no]fastmd` option. This option is described in subsection 2.3.27.19, page 29.

†† Cray Research extension (Cray Standard C only)

Section 3

Characters

3.1.2.3

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 at sign (`@`) and dollar sign (`$`) in identifier names. For more information on the `-h [no]calchars` option, see subsection 2.3.27.10, page 25.

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 char` type. Cray C++ does not distinguish between a plain `char` type and an unsigned `char` for the purpose of overloaded function resolution.

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 example:

Character constant	Integer value
`'a'`	`0x61`
`'ab'`	`0x6162`

In a character constant or string literal, if an escape sequence is not recognized, the backslash character (\) that initiates the escape sequence is ignored, as shown in the following example:

Character constant	Integer value	Explanation
`'\a'`	`0x7`	Recognized as the BEL character
`'\8'`	`0x8`	Not recognized
`'\['`	`0x5b`	Not recognized; ASCII value for `[`
`'\c'`	`0x63`	Not recognized; ASCII value for `c`

Wide characters
3.1.2.4

Wide characters are treated as signed 64-bit `int` 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`(3) 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.

Note: The Cray C++ 2.0 compiler does not support `wchar_t` as a separate type.

Integers
3.1.2.5

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 2, page 61.

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 Cray PVP systems and 1's on Cray MPP systems. On Cray MPP systems, this behavior can be modified by using the `-h nosignedshifts` option (see subsection 2.3.27.45, page 43)

On Cray MPP systems, the C 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 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 Cray PVP 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.

Floating-point arithmetic
3.1.2.6

Cray Research systems use either Cray floating-point arithmetic or IEEE floating-point arithmetic. These types of floating-point representation are described in the subsections that follow.

Cray floating-point representation
3.1.2.6.1

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:

$$2.7 \times 10^{2465}$$

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:

$$3.67 \times 10^{-2466}$$

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`.

Rounding of floating-point arithmetic is toward 0. 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:

$$\textit{number of decimal digits} = \frac{\textit{number of bits}}{\log_2(10.0)}$$

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`.

IEEE floating-point representation
3.1.2.6.2

On Cray MPP systems, `float` represents IEEE single-precision (32-bit) floating-point values; `double` and `long double` represent double-precision (64-bit) floating-point values.

On Cray PVP 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 Cray PVP systems.

Section 3

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.

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 floating-point arithmetic is determined by the current 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:

$$\textit{number of decimal digits} = \frac{\textit{number of bits}}{\log_2(10.0)}$$

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 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 using the `cpu`(8) command (see the `cpu`(8) man page for more information).

Arrays and pointers

3.1.2.7

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 in 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 equally.

Pointers on Cray PVP systems differ from pointers on Cray MPP systems. The subsections that follow describe pointer implementation on each type of system.

Pointers on Cray PVP systems

3.1.2.7.1

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 Cray MPP 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.

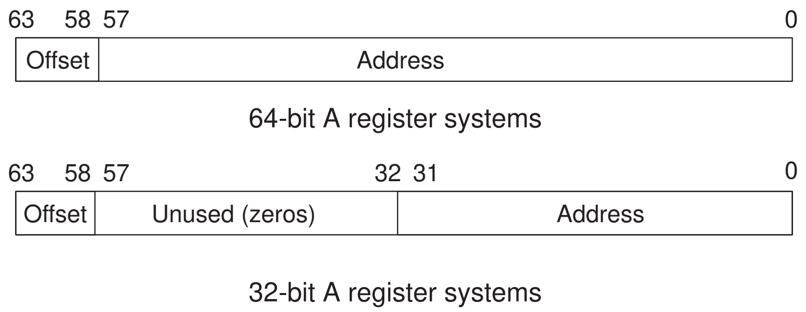

Figure 1. Character pointer format

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.

Pointers on Cray MPP systems

3.1.2.7.2

Pointers on Cray MPP 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.

Registers

3.1.2.8

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. It is an error to take the address of a variable (using the address operator, `&`) declared with `register` storage class. This restriction is useful for optimization because aliasing conflicts are not allowed.

Classes, structures, unions, enumerations, and bit fields

3.1.2.9

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.

A member of a class or structure (`struct`) that is a bit-field or that is of type `char` is termed a *part-word member*. On Cray MPP systems, `float` and `short` types are also part-word members. All data members of classes or structures other than part-word members are aligned on word boundaries. Part-word members are packed into Cray words from left to right. Part-word members do not straddle a word boundary. Padding to the next word boundary is appended to a part-word member only if the following member is not a part-word member or would otherwise straddle a word boundary.

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 programs; they are a separate type in C++ programs.

Section 3

Qualifiers

3.1.2.10 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.

Declarators

3.1.2.11 C allows a maximum of 12 pointer, array, and/or function declarators to modify an arithmetic, structure, or union type. C++ has no fixed limit. Environmental limits are listed in Table 3, page 72.

Statements

3.1.2.12 C allows a `switch` statement to have a maximum of 4095 case values. C++ has no fixed limit. Environmental limits are listed in Table 3, page 72.

The C++ compiler parses `asm` statements for correct syntax, but otherwise ignores them.

Exceptions

3.1.2.13 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.

System function calls

3.1.2.14 See the `exit`(1) man page for a description of the form of the unsuccessful termination status that is returned from a call to `exit`.

If the attempt is successful, a call to operator `new` returns a pointer to the allocated storage. Otherwise, if the last argument to `set_new_handler` was a null pointer, operator `new` returns a `NULL`.

Preprocessing

3.1.3

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 subsection 2.3.9, page 12.

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 section 6, page 129.

The `#pragma` directives are described in section 4, page 75.

Environmental limits

3.1.4

To conform with the C standard, a compiler must be able to translate and execute programs that contain a specified set of environmental limits. Table 3, following, describes these limits, their values as defined in the C standard, and their values as defined in Cray Standard C. Dependences exist among various limits.

> **Note:** Unlike C, the C++ standard does not require specific environmental limits.

Table 3. Standard C environment limits

Description	Standard C limit	Cray Standard C limit
Nesting levels of compound statements, iteration control structures, and selection control structures	15	100
Nesting levels of conditional inclusion	8	65535
Pointer , array, and function declarators (in any combination) modifying an arithmetic type, a structure, a union, or an incomplete type in a declaration	12	12
Nesting levels of declarators enclosed in parentheses within a full declarator	31	100
Nesting levels of expressions enclosed in parentheses within a full expression	32	65535
Significant initial characters in an internal identifier or a macro name	31	255
Significant initial characters in an external identifier	6	255
External identifiers in one translation unit	511	1020
Identifiers with block scope declared in one block	127	4000
Macro identifiers simultaneously defined in one translation unit	1024	65499 †
Parameters in one function definition	31	254
Arguments in one function call	31	254

† Includes predefined macros

Section 3

Table 3. Standard C environment limits (continued)

Description	Standard C limit	Cray Standard C limit
Parameters in one macro definition	31	31
Arguments in one macro invocation	31	31
Characters in a logical source line	509	††
Characters in a character string literal or wide string literal (after concatenation)	509	65528 †††
Bytes in an object	32767	2^{35}–8 ††††
Nesting levels for `#include` files	8	65534
Case labels for a `switch` statement (excluding those for any nested `switch` statements)	257	4095
Members in a single structure or union	127	50000
Enumeration constants in a single enumeration	127	50000
Levels of nested structure or union definitions in a single `struct-declaration-list`	15	65535

†† Limited only by the size of the source file and by the amount of memory available on the machine.

††† Limit shown is for normal characters. Wide character strings have a limit of 8191.

†††† Limit for a hosted environment only. This is the theoretical limit; the actual limit is set by the amount of memory available on the machine.

#pragma Directives [4]

The #pragma directives are extensions to the C and C++ standards. They are classified according to the following types:

- General
- Template instantiation (Cray C++ only)
- Vectorization
- Scalar
- Tasking
- Inlining

You can control CRI-specific compiler analysis of various aspects of your source code by using #pragma directives. #pragma directives have the following form:

#pragma [_CRI] *identifier* [*arguments*]

In the specification, macro expansion is applied only to *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 any directives that do not contain the _CRI specification.

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

Macro expansion occurs on the directive line after the directive name. (That is, macro expansion occurs after the *identifier* .) 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)
```

Section 4

The expanded code is equivalent to the following:

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

At the beginning of each subsection that describes a directive, information is included about the compilers (C and/or C++) and systems that allow the use of the directive (PVP and/or MPP), and the scope (local and/or global) of the directive. Unless otherwise noted, the following default information applies to each option:

Compiler:	C and C++
Systems:	PVP and MPP
Scope:	Local and global

Protecting directives

4.1

To ensure that your directives are seen 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.

Section 4

Directives in Cray C++

4.2

The C++ programming language strictly 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 the C programming language.

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 instantiate directive as described in subsection 5.3, page 125.

4.3 General directives

General directives specify compiler actions that are specific to the directive and have no similarities to the other types of directives.

4.3.1 [no]bounds directive

Compiler: C only

You can use either a command-line option or a directive, or both, to specify bounds checking. 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.

The bounds directive specifies that pointer and array references are to be checked. The nobounds directive specifies that this checking is to be disabled. 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 [no]bounds directive:

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

duplicate directive
4.3.2

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.

Section 4

The following example illustrates the use of the `duplicate` directive:

```
#include <complex.h>

extern void maxhits(void);

#pragma _CRI duplicate maxhits as count, quantity     /* OK */

void maxhits(void)
{
   #pragma _CRI duplicate maxhits as tempcount
   /* Error:  #pragma _CRI duplicate cannot appear in local scope */
}

double complex minhits;

#pragma _CRI duplicate minhits as lower_limit
/* Error:  minhits is not declared as a function */

extern void derivspeed(void);

#pragma _CRI duplicate derivspeed as accel
/* Error:  derivspeed is not defined in this file */

static void endtime(void)
{
}

#pragma _CRI duplicate endtime as limit
/* Error:  endtime is defined as a static function */
```


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.

The following example references duplicate names:

```
void converter(void)
{
   structured(void);
}

#pragma _CRI duplicate Converter as factor, multiplier /* OK */

#pragma _CRI duplicate remainder as left, unused /* OK */

void remainder(void)
{
}

#pragma _CRI duplicate remainder as available         /* OK */

#pragma _CRI duplicate remainder as factor, structured, left
/* Error:  factor, structured, and left are referenced in this file */
```

Duplicate names can be used to provide alternate external names for functions, as the following examples show.

main.c:

```
extern void fctn(void), FCTN(void);

main()
{
  fctn();
  FCTN();
}
```

fctn.c:

```
#include <stdio.h>

#pragma _CRI duplicate fctn as FCTN
void fctn(void)
{
  printf("Hello world\n");
}
```

Files main.c and fctn.c are compiled and linked using the following command line:

```
cc main.c fctn.c
```

When the executable file a.out is run, the program generates the following output:

```
Hello world
Hello world
```

message directive

4.3.3

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"
```

Section 4

The following example illustrates the use of the `message` compiler directive:

```
#define FLAG 1

sub() {

#ifdef FLAG
#pragma _CRI message "FLAG is Set"
#else
#pragma _CRI message "FLAG is NOT Set"
#endif

}
```

[no]opt directive
4.3.4

Scope: Global only

You can control optimization of your source code by means of either command-line options or directives, or both. Disabling optimization removes various sources of potential confusion in debugging, and leaves the program order found in the source.

The `opt` directive directs the compiler to optimize the source code that follows the directive. The `noopt` directive disables all automatic optimizations and causes optimization directives to be ignored. 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
```

The following example illustrates the use of the opt and noopt compiler directives:

```
#include <stdio.h>

void sub1(void) {
        printf("In sub1, default optimization\n");
}

#pragma _CRI noopt
void sub2(void) {
        printf("In sub2, optimization disabled\n");
}
#pragma _CRI opt

void sub3(void) {
        printf("In sub3, optimization enabled\n");
}

main() {
        printf("Start main\n");
        sub1();
        sub2();
        sub3();
}
```


soft directive
4.3.5

Scope: Global only

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 (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.

The `soft` directive must appear at the global scope level. In C, it can appear before or after the declaration of any of the named functions and objects. In C++ it must appear after the declarations of all named objects or functions. 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.

The following example illustrates these restrictions:

```
extern long x;          /* x is a soft external data object */
extern void f(void);    /* f is a soft external function */

extern long y;
#pragma _CRI soft x, y, f
long y = 4;             /* ERROR - y is actually defined */

static long z;

#pragma _CRI soft z     /* ERROR - z is declared static */

void fctn(void) {
#pragma _CRI soft a     /* ERROR - directive must be at global scope */
}
```

> **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 the user's 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.

4.3.6 `vfunction` directive

Systems: PVP only

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.

The following example illustrates the use of the `vfunction` compiler directive:

```
#pragma _CRI vfunction vf
extern double vf(double);

void f3(int n) {
    int i;
    for (i=0; i<n; i++) {     /* Vectorized */
        b[i] = vf(c[i]);
    }
}
```

Section 4

4.4 Instantiation directives

The Cray C++ compiler recognizes several instantiation directives. Instantiation directives can be used to control the instantiation of specific template entities or sets of template entities. The following directives are described in detail in subsection 5.3, page 125:

- `#pragma instantiate`
- `#pragma do_not_instantiate`
- `#pragma can_instantiate`

4.5 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 Code on Cray PVP Systems*, publication SG–2192.

The subsections that follow describe the compiler directives used to control vectorization. These directives are not available on Cray MPP systems.

Section 4

4.5.1 `ivdep` directive

Systems: PVP only
Scope: Local only

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
```

The following example illustrates the use of the `ivdep` compiler directive:

```
p = a; q = b;
#pragma _CRI ivdep
for (i=0; i<n; i++) {         /* Vectorized */
         *p++ = *q++;
}
```

`novector` directive
4.5.2

Systems: PVP only
Scope: Local only

The `novector` directive directs the compiler not to 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
```

The following example illustrates the use of the `novector` compiler directive:

```
#pragma _CRI novector
for (i=0; i<h; i++) {    /* Loop not vectorized */
    a[i] = b[i] + c[i];
}
```

novsearch directive
4.5.3

Systems: PVP only
Scope: Local only

The novsearch directive directs the compiler not to 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
```

The following example illustrates the use of the novsearch compiler directive:

```
#pragma _CRI novsearch
for (i=0; i<h; i++) {    /* Loop not vectorized */
    if (a[i] < b[i]) break;
    a[i] = b[i];
}
```

Section 4

prefervector directive
4.5.4

Compiler: C (release 5.0 only), C++
Systems: PVP only
Scope: Local only

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
```

The following example illustrates the use of the `prefervector` directive:

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

In the preceding example, both loops can be vectorized, but the directive directs the compiler to vectorize the outer `for` loop. Without the directive and without any knowledge of `n` and `m`, the compiler vectorizes the inner `for` loop. In this example, the outer `for` loop is vectorized even though the inner `for` loop had a `#pragma _CRI ivdep` directive. See also the `#pragma _CRI prefertask` directive in subsection 4.6.7, page 101.

`shortloop` directive
4.5.5

Systems: PVP only
Scope: Local only

The `shortloop` directive improves 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 format of this directive is as follows:

```
#pragma _CRI shortloop
```

Section 4

The following example illustrates the use of the shortloop directive:

```
#pragma _CRI shortloop
for (i=0; i<n; i++) {   /* 1<=n<=64 */
    a[i] = b[i] + c[i];
}
```

Tasking directives

4.6

On Cray PVP systems using multiple processors, the Cray Standard C compiler and the Cray C++ compiler support *parallel processing*. 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 Code on Cray PVP Systems*, publication SG–2192, 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 automates loop-level tasking by detecting parallelism in a program and exploiting the parallelism without user intervention. You can add directives to your code that identify loops you know can be tasked, and depend upon Autotasking to analyze other loops.

  Generally , Autotasking works best on programs in which most of the work is in nested loops that do not contain function calls.

  Autotasking is enabled by specifying the -h task*n* option on the command line. For more information on the -h task*n* option, see subsection 2.3.27.50, page 45.

Section 4

- *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.

  Those jobs that do significant amounts of work and are inherently parallel are candidates for tasking. You must determine where to insert the tasking directives, determine the tasking context of variables, and check whether the results are correct.

The autotasking directives described in the following subsections are not available on Cray MPP systems. They are available on Cray PVP systems running Cray Standard C or Cray C++.

4.6.1 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.

4.6.1.1 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.

Argument	Description
`private`	The `private` argument indicates those variables in the parallel region that are to have private context.
`shared`	The `shared` argument indicates those variables in the parallel region that are to have shared context.
`value`	The `value` argument indicates those variables in the parallel region that are to have value context; that is, each task has its own initialized private copy of these variables.
`defaults`	The `defaults` argument indicates that all variables referenced in the parallel region but not listed in the `private`, `shared`, or `value` lists are to have their tasking context determined by using the default rules. The `private`, `shared`, and `value` arguments can be used in conjunction with the `defaults` argument to override the default rules for specified variables

Work distribution arguments 4.6.1.2

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.

Argument	Description
`single`	The `single` argument specifies that the iterations are distributed one at a time to available processors. If no work distribution arguments are specified, the default is `single`.
`chunksize (`*exp*`)`	The `chunksize` argument specifies that the iteration space is broken into chunks of size *exp*, with a possible smaller residual chunk. *exp* is an integral expression enclosed in parentheses following the `chunksize` argument. This expression is evaluated at run time, just prior to execution of the loop. If `chunksize` is less than or equal to 0, the `chunksize` is assumed to be 1. Otherwise, if the `chunksize` value is greater than the number of iterations in the loop, the value is set to the number of iterations. The `chunksize` expression cannot contain side effects.
`numchunks (`*exp*`)`	The `numchunks` argument specifies that the iteration space is broken into *exp* chunks of equal size, with a possible smaller residual chunk. *exp* is an integral expression enclosed in parentheses following the `numchunks` argument. This expression is evaluated at run time, just prior to execution of the loop. If the `numchunks` expression value is less than or equal to 0, the value is set to 1. If the `numchunks` expression value is greater than or equal to the number of iterations, the value is set to the number of iterations. The `numchunks` expression cannot contain side effects.

Argument	Description
`guided`	The `guided` argument splits the iteration space into chunks of varying sizes. The sizes of the chunks are computed at run time by using an algorithm that attempts to balance the load on each processor so that all processors finish their work at approximately the same time.
`vector`	The `vector` argument splits the iteration space into chunks of varying sizes down to a minimum size of 64 or 128, depending on the vector length of the target machine. The sizes of the chunks are computed at run time by using an algorithm that attempts to balance the load on each processor so that all processors finish their work at approximately the same time.

Miscellaneous arguments
4.6.1.3

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

Argument	Description
`if (`*exp*`)`	The scalar expression (*exp*) enclosed in parentheses that follows the `if` argument is evaluated at run time, just prior to execution of the parallel region. If *exp* is true, tasking is performed. If *exp* is false, tasking is not performed. The `if` expression cannot contain side effects. This argument can be used only with the `parallel` directive or `taskloop` directive for stand-alone task loops.

Argument	Description
`maxcpus (`*exp*`)`	The `maxcpus` argument specifies the maximum number of CPUs that the parallel region can effectively use. Specifying `maxcpus` does not ensure that *exp* processors will be assigned; it simply specifies the optimal maximum. If *exp* is less than or equal to 0, it is assumed to be 1.
`savelast`	The `savelast` argument indicates that the final iteration of the loop (or loop partition, when used in conjunction with the `vector` or `chunksize` arguments) will be performed by the master processor after all the other iterations (or partitions) have been completed. This argument ensures that all `private` and `value` variables with values that are set inside the loop will have the values produced by the final loop iteration when the loop is exited; this is not ensured by default. The `savelast` argument has no parameters. This argument can be used only with stand-alone task loops.

`case` and `endcase` directives
4.6.2

Systems: PVP only
Scope: Local only

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
```

In the following example, the first, second, and third loops execute concurrently:

```
#pragma _CRI parallel private(i) shared(a,b,c)
#pragma _CRI case
        for (i=0; i<1000; i++) {
        /* This loop #1 executes in parallel with loops
            #2 and #3. */
           a[i] = 0;
        }
#pragma _CRI case
        /* This loop #2 executes in parallel with loops
            #1 and #3. */
        for (i=0; i<1000; i++) {
           b[i] = 1;
        }
#pragma _CRI case
         /* This loop #3 executes in parallel with
            loops #1 and #2. */
        for (i=0; i<1000; i++) {
           c[i] = 2;
        }
#pragma _CRI case            /* empty case is allowed */
#pragma _CRI endcase
        .
        .
        .
#pragma _CRI endparallel
```

common directive
4.6.3

Systems: PVP only

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 command-line option. The format of the common directive is as follows:

```
#pragma _CRI common variables...
```

The same variable can be named in both the common and the taskcommon directive, and in fact multiple copies of each directive naming the same variable are allowed; in these cases the last directive takes precedence. The common directive can appear in both global and local scopes and can be applied to both static and nonstatic local variables.

In C, the common directive can appear before or after the declaration of the named objects. In C++ it must appear after the declaration of all named objects.

The following example illustrates the use of the common directive:

```
/* The #pragma directive below retains "test" in
   common storage when the -h taskcommon command-line
   option is used. */
int test;
#pragma _CRI common test  /* Keep "test" in common */
```

Section 4

guard and endguard directives
4.6.4

Systems: PVP only
Scope: Local only

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.

The following example illustrates the use of the `guard` and `endguard` directives:

```
#pragma _CRI guard     /* protect the update of sum and big */
      sum = sum + xsum;
      big = max(xbig, big);
#pragma _CRI endguard
```

endloop directive
4.6.5

Systems: PVP only
Scope: Local only

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
```

In the following example, a parallel region is defined that uses a `taskloop/endloop` pair and a guarded region to implement a reduction computation.

```
sum = 0;
      big = -1;
#pragma _CRI parallel private(i,xsum,xbig) shared       \
(aa,bb,cc,sum,big)
      xsum = 0;
      xbig = -1;
#pragma _CRI taskloop vector
      for (i=0; i<2000; i++) {
        ...
        xsum = xsum + aa[i]*(bb[i]-cc[aa[i]]);
        xbig = max(abs(aa[i]*bb[i], xbig);
         ...
     }
#pragma _CRI guard/* protect the update of sum and big */
  sum = sum + xsum;
      big = max(xbig,big);
#pragma _CRI endguard
#pragma _CRI endloop

        ...
    /* ensure that all processors have contributed to */
    /* the sum; all processors are held here until    */
    /* all contributions are in, ensuring that the    */
    /* value of sum and big will be correct for their */
    /* later use within the parallel region.          */
         ...
       if (sum > 1000.) {
         ...
 #pragma _CRI endparallel
```

Section 4

In this example, the guarded region protects the update of `sum` and `big`, so that each processor does its own update without interference from the others. (Guarded regions are discussed in subsection 4.6.4, page 97.) The `endloop` directive ensures that no processor can proceed beyond this point until all have contributed to the `sum` and `big` values.

parallel and endparallel directives
4.6.6

Systems: PVP only
Scope: Local only

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 subsection 4.6.1, page 91.

prefertask directive
4.6.7

Compiler: C (release 5.0 only), C++
Systems: PVP only

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 and 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
```

The following example illustrates the use of the `prefertask` directive:

```
for (i=0; i<n; i++)
{
   #pragma _CRI prefertask
   for (j=0; j<m; j++)
      e[j][i] = f[j][i] + g[j][i];
}
```

In the preceding example, both loops can be tasked, but the directive directs the compiler to task the inner `for` loop. Without the directive and without any knowledge of `n` and `m`, the compiler tasks the outer `for` loop. Using the directive, the loops are interchanged (to increase parallel granularity) and the resulting outer `for` loop (involving `j`) is tasked. See also the `prefervector` directive in subsection 4.5.4, page 88.

4.6.8 `taskcommon` directive

Systems: PVP only

You can specify the task common storage class for variables by naming them in the `taskcommon` directive. The format of this directive is as follows:

```
#pragma _CRI taskcommon (variables )
```

The list of variables can be enclosed in parentheses. This directive applies only to the following types of variables:

- Global scope variables, in which case the directive must appear at global scope.
- Static local scope variables, in which case the directive must appear within the same scope as the variable declaration, and the variable must have a value assigned to it somewhere in the program.

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

In C, the `taskcommon` directive can appear before or after the declaration of the named objects. In C++, it must appear after the declaration of all named objects.

The following example illustrates the use of the `taskcommon` compiler directive:

```
int x;
#pragma _CRI taskcommon x
fctn()
{
 extern int x;
}
```

In the preceding example, the interior declaration of `x` as `extern int` is the same `x` previously declared to be task common and has the task common storage class.

The `taskcommon` directive takes precedence over the `-h common` command-line option.

4.6.9 `taskloop` directive

Systems: PVP only
Scope: Local only

The `taskloop` directive indicates that the following loop can be executed in parallel by multiple processors. No directive is needed to end a `taskloop` 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 subsection 4.6.1, page 91.

The following example illustrates the use of the `taskloop` directive:

```
#pragma _CRI taskloop vector
for (i=0; i<2000; i++) {
    ...
    xsum = xsum + aa[i]*(bb[i]-cc[aa[i]]);
    xbig = max(abs(aa[i]*bb[i], xbig);
    ...
}
```


Scalar directives

4.7

Scalar optimization directives control aspects of code generation, register storage, and bottom loading. The subsections that follow describe these directives.

`align` directive
4.7.1

Systems: PVP only

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.

Function alignment
4.7.1.1

When the `align` directive appears in global scope, it must contain a list, optionally parenthesized, of functions that are to be aligned. Each function in the list must be defined somewhere in the file. The code generated for a function that has been specified on a valid, global `align` directive will be aligned on a machine instruction buffer boundary.

For function alignment, the `align` directive appears in global scope and has the following format:

```
#pragma _CRI align func...
```

Section 4

The following example illustrates the use of the `align` directive when it is used in global scope for function alignment:

```
void mutt();
#pragma _CRI align mutt, hound   /* Error:  mutt and hound are declared
                                            but not defined in this file */
void hound();

#pragma _CRI align siamese       /* Error:  siamese not defined in the file */
extern long dachshund;
#pragma _CRI align dachshund     /* Error:  dachshund has already been defined
                                            and its type is not function */
#pragma _CRI align shih_tzu      /* Error:  shih_tzu is defined as a short */
short shih_tzu;

#pragma _CRI align poodle        /* OK */
poodle()
{
}

double chow();
#pragma _CRI align chow          /* OK */
double chow()
{
}

dalmatian()
{
}
#pragma _CRI align dalmatian     /* OK */
#pragma _CRI align dalmatian     /* OK, but unnecessary duplication */
```

The `align` directive is especially useful for small functions that fit entirely into the machine's instruction buffers. By using this directive, you can ensure that such functions are executed entirely within the instruction buffers available on the machine. After the executable code for the function has been read into the instruction buffers, no additional instruction buffer fetches will be necessary while the function executes.

The code resulting from an inlined function is not aligned, even if the function' s name is specified on an `align` directive. The performance improvement obtained by aligning code on an instruction buffer is generally insignificant when compared to that obtained by inlining.

Section 4

In C, the `align` directive can appear before or after the declaration of any of the named functions. In C++, it must appear after the declarations of all the named functions.

The `-h align` option can be used to specify that all functions defined in the file are to be aligned automatically on instruction buffer boundaries.

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*, publication SR–2011.

Loop alignment
4.7.1.2

The compiler automatically aligns `for`, `while`, and `do...while` loops when it determines that this will increase performance. It aligns any loops that fit into and require all of the machine's instruction buffers, because the best performance improvement results from executing a loop entirely from within the buffers.

You can specify alignment of a particular loop by using the `align` directive. For loop alignment, the `align` directive must appear in local scope and has the following form:

```
#pragma _CRI align
```

The `align` directive is especially useful for small loops that fit entirely into the machine's instruction buffers. By using this directive, you can ensure that such loops are executed entirely within the instruction buffers available on the machine. After the executable code for the loop has been read into the instruction buffers, no additional instruction buffer fetches are necessary while the loop executes.

To determine the number and size of the instruction buffers on your system, use the `target`(1) command which is described in the *UNICOS User Commands Reference Manual*, publication SR–2011.

The compiler does not automatically align any of the following kinds of loops:

- `if...goto` loops
- Loops containing function calls, unless all function calls have been inlined

In addition, automatic loop aligning is disabled when the `-h scalar0` option is specified.

Because the compiler automatically aligns loops, for most programs you will never need to use the `align` directive. However, at times it is desirable to align loops in addition to those automatically aligned by the compiler. In these cases, you must use the `align` directive.

The `align` directive must appear directly preceding a `for`, `while`, or `do` statement or a label; it must not appear in any other context within local scope. (As described in the previous subsection, the `align` directive may appear in global scope to indicate functions to be aligned.)

The code generated for the statement that follows the `align` directive is aligned on the next instruction buffer boundary. An `align` directive directly preceding a `for` loop causes code starting at the second expression of the `for` statement to be aligned on an instruction buffer boundary. When any code space is left unused, either it is filled with `NOP` instructions or a `jump` instruction is generated, whichever executes faster.

Use of the `align` directive before a label lets you align `if...goto` loops, where the label designates the top of the loop that is the target of the `goto` statement. However, use of the directive for labels is not restricted to this use. Any referenced label that is preceded by a `align` directive will be aligned on an instruction buffer boundary. If the label is never referenced, the `align` directive has no effect.

If a loop following an `align` directive is tasked, the directive applies only to the loop in the unitasked section of code. This is because the original loop does not appear in its original form in the master code and slave function. Any loops that are not tasked, and labels following an `align` directive in a parallel region, are aligned in all three versions of the generated code (master , slave, and unitasked).

Generally , the `align` directive has no effect on automatically aligned loops. However , if a loop is aligned by using the `align` directive, the resulting increase in code size may prevent alignment of an outer loop that would normally have been automatically aligned because the outer loop no longer fits within the instruction buffers. Conversely, an outer loop that would have been too small to be aligned, may now be automatically aligned because of the increased size of an aligned inner loop.

The following example illustrates the use of the `align` directive when it is used in local scope for loop alignment:

```
void fctn()
{
   int i;
#pragma _CRI align
   for (i=0; i<100; i++) /* This loop will be aligned. */

   {
      .
      .
      .
   }
   i = 0;
#pragma _CRI align
   top_of_loop:  ; /* This loop will be aligned. */
      .
      .
      .
      i++;
   if (i < 100) goto top_of_loop;
}
```

cache_align directive
4.7.2

Systems: MPP only

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.

In C, the cache_align directive can appear before or after the declaration of the named objects. In C++, it must appear after the declaration of all named objects.

Section 4

noreduction directive
4.7.3

Scope: Local only

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

```
#pragma _CRI noreduction
```

The following example illustrates the use of the noreduction compiler directive:

```
sum = 0;
#pragma _CRI noreduction
for (i=0; i<n; i++) {
    sum += a[i];
}
```

split directive
4.7.4

Systems: CRAY T3E only
Scope: Local only

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 CRAY T3E 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 forms:

```
#pragma _CRI split
```

The split compiler directive should be placed immediately before the loop to be split. It should immediately precede a for, while, do or label statement. It should not appear in any other context.

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.

For example:

```
#pragma _CRI split
      for (i=0; i<1000; i++) {
         a[i] = b[i] * c[i];
         t = d[i] + a[i];
         e[i] = f[i] + t * g[i];
         h[i] = h[i] + e[i];
      }
```

First, the compiler generates the following loop (notice the expansion of the scalar temporary `t` into the compiler temporary array `ta`):

```
for (i=0; i<1000; i++) {
   a[i] = b[i] * c[i];
   ta[i] = d[i] + a[i];
}
for (i=0; i<1000; i++) {
   e[i] = f[i] * ta[i] * g[i];
   h[i] = h[i] + e[i];
}
```

Finally , the compiler stripmines the loops to increase the potential for cache hits and reduce the size of arrays created for scalar expansion:

```
for (i1=0; i1<1000; i1+=256) {
   i2 = (i1+256 < 1000) ? i1+256 : 1000;
   for (i=i1; i<i2; i++) {
      a[i] = b[i] * c[i]
      ta[i-i1] = d[i] + a[i]
   }
   for (i=i1; i<i2; i++) {
      e[i] = f[i] * ta[i-i1] * g[i]
      h[i] = h[i] + e[i]
   }
}
```

suppress directive
4.7.5

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 form:

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

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

In C, the suppress directive can appear before or after the declaration of the named objects or functions. In C++, it must appear after the declaration of all the named functions or objects.

For Cray MPP 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.

symmetric directive
4.7.6

Compiler: C++ only
Systems: MPP only
Scope: Local only

The #pragma 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 #pragma 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.

[no]unroll directive
4.7.7

Scope: Local only

The unrolling directives allow 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]
```

n specifies the total number of loop body copies to be generated. *n* must be in the range of 2 through 64.

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.

Warning: 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 whose iteration counts 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. For more information on the `-h [no]unroll` option, see subsection 2.3.27.56, page 48.

On Cray MPP 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 Cray PVP systems, the compiler may do additional unrolling over the amount requested by the user.

Section 4

In the following example, assume that the outer loop of the following nest will be unrolled by two:

```
#pragma _CRI unroll 2
for (i=0; i<10; i++) {
      for (j=0; j<100; j++) {
        a[i][j] = b[i][j] + 1;
      }
    }
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent to each other:

```
for (i=0; i<10; i+=2) {
      for (j=0; j<100; j++) {
        a[i][j] = b[i][j] + 1;
      }
      for (j=0; j<100; j++) {
        a[i+1][j] = b[i+1][j] + 1;
      }
    }
```

The compiler then jams the inner two loop bodies together, producing the following nest:

```
for (i=0; i<10; i+=2) {
      for (j=0; j<100; j++) {
        a[i][j] = b[i][j] + 1;
        a[i+1][j] = b[i+1][j] + 1;
      }
    }
```

Outer loop unrolling is not always legal because the transformation can change the semantics of the original program. For example, unrolling the following loop nest on the outer loop would change the program semantics because of the dependency between a[i][...] and a[i+1][...]:

```
/*
 * Directive will cause incorrect code due to dependencies!*/
#pragma _CRI unroll 2
for (i=0; i<10; i++) {
      for (j=1; j<100; j++) {
        a[i][j] = a[i+1][j-1] + 1;
      }
    }
```

Inlining directives

4.8

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 subsection 2.3.27.26, page 31.
- Inlining of particular function calls is specified by the `inline` directive, as discussed in the subsections that follow.

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.

`inline` directive
4.8.1

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

```
#pragma _CRI inline (func.. .)
```

func... represents the function or functions to be inlined. The list can be enclosed in parentheses. Listed functions must be defined in the compilation unit. In C, the `inline` directive can appear before or after the declaration of any of the named functions. In C++, it must appear after the declarations of all the named functions. You cannot specify objects of type pointer -to-function.

The following example illustrates the use of the `inline` directive:

```
#include <stdio.h>
int f(int a) {
  return a*a;
}
#pragma _CRI inline f       /* Direct the compiler to inline
                               calls to f. */
main() {
  int b = 5;
  printf("%d\n", f(b));     /*  Call of f is inlined here */
}
```

Section 4

noinline directive
4.8.2

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

```
#pragma _CRI noinline (func.. .)
```

func... 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. In C, the `noinline` directive can appear before or after the declaration of any of the named functions. In C++, it must appear after the declarations of all the named functions. You cannot specify objects of type pointer-to-function.

The following example illustrates the use of the `noinline` directive:

```
#include <stdio.h>
int f(int a) {
  return a*a;
}
#pragma _CRI noinline f    /* Direct the compiler not to
                              inline calls of f. */
main() {
  int b = 5;
  printf("%d\n", f(b));    /* Call of f is not inlined here */
}
```

Template Instantiation [5]

The Cray C++ 2.0 compiler supports templates. 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 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 by the C++ language. 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.
- The C++ language also dictates that 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.

Section 5

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.

Automatic instantiation

5.1

The goal of an automatic instantiation mode is to provide painless 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 by using implicit inclusion (see subsection 5.4, page 128).

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 object files that are generated contain information about entities that are candidates for instantiation.

2. When the object files are linked together, the prelinker program examines the object files, looking for references and definitions of template entities and for any additional information about entities that are candidates for instantiation.

3. If the prelinker finds a reference to a template entity that has no definition in the set of object files, it looks for a file that indicates instantiation of 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. Each file that causes the associated `.ii` file to be changed is then recompiled.

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.

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, finds all necessary instantiations present in the object files, and adjusts instantiation assignment only if the set of required instantiations changes. Since 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.

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 within one file is changed in such a way that some instantiation within that file no longer compiles. If a specialization is added to another file and the first file is recompiled before the specialization file, errors are generated. 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 command-line specification of the instantiation mode.

The automatic instantiation mode can be disabled by issuing the `-h noautoinstantiate` command-line option. If automatic instantiation is disabled, the information about template entities within a source file that are candidates for instantiation are 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.

Section 5

mode	Description
`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 `-h instantiate=used`, 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 `-h instantiate=used` mode is used to suppress automatic instantiation.

Instantiation `#pragma` directives

5.3

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.

Section 5

- 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 member function name. For example: A<int>::f
- A static data member name. For example: 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>) 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 #pragma 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 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 exists
#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 `#pragma` directive cannot be a compiler -generated function, an inline function, or a pure virtual function.

Implicit inclusion

5.4

The implicit inclusion feature is enabled by default. This implies that the front end has permission to assume that if it 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 alongside 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]

Section 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 subsection 2.3.27.15, page 27).

> **Note:** Any of the predefined macros except those required by the standard (see subsection 6.1) 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 Systems Libraries Reference Manual*, publication SR–2080.

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.

Macro	Description
`__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.

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.
`_UNICOS`	Defined as the integer portion of the major release level of the current UNICOS release (for example, 8 or 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.
`CRAY`††	Defined as 1 on all Cray Research systems.
`_CRAY`	Defined as 1 on all Cray Research systems.

† The `__cplusplus` macro is required by the ISO C++ draft standard, bu ISO C standard.

†† This macro is not defined in strict-conformance mode.

Macro	Description
`CRAY1`[††]	Defined as 1 on all Cray PVP systems; if the hardware is any other machine type, the macro is not defined.
`_CRAY1`	Defined as 1 on all Cray PVP systems; if the hardware is any other machine type, the macro is not defined.
`_CRAYMPP`	Defined as 1 on all Cray MPP 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; undefined when Cray format is used.
`_ADDR32`	Defined as 1 if the targeted CPU has 32-bit address registers.
`_ADDR64`	Defined as 1 if the targeted CPU has 64-bit address registers.
`_LD64`	Defined as 1 if the `long double` basic type has 64 bits of precision; undefined if 128-bit precision is used.
`_FASTMD`	Defined as 1 if the fast multiply/divide sequence is enabled. It is not defined if the machine type is MPP.
`_MAXVL`	Defined as the maximum hardware vector length (64 or 128). It is not defined if the machine type is MPP.

†† This macro is not defined in strict-conformance mode.

Macros based on the compiler

6.4

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.

Intrinsic Functions [7]

Section 7

The set of 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, and no calls to external functions are generated. 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.

Unless otherwise stated, arguments to these functions must be of type `short`, `int`, or `long`, or their signed or unsigned varieties. The types of the arguments 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.

Where constant values are required for arguments, constant expressions are also allowed.

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.

Arguments to these functions that specify bit position are numbered from the right, with the rightmost bit in bit position 0. For example, in the following set of bits, the only bit set is bit 4.

```
00000000000000000000000000010000
```

For detailed descriptions of appropriate Cray Research hardware instructions, see the *Symbolic Machine Instructions Reference Manual*, publication SR–0085.

For your convenience, a summary is provided at the end of this section, listing all intrinsic functions in alphabetical order.

Intrinsics for all Cray Research systems

7.1

The following intrinsic functions are available on all Cray Research computer systems.

Function	Description
`int _argcount()` (Not available in Cray Standard C 4.0)	The `_argcount` function returns the number of arguments passed to the function in which the call appears. If `_numargs` is called from within an inline function, the return value is undefined. For this reason this intrinsic function should never be called from within a file compiled with an automatic inlining level greater than 0. (See `-h inline`*n* in subsection 2.3.27.26, page 31, and the `_argcount` and `_numargs` intrinsic functions for more information.) When this intrinsic function is called from within a function that returns a `struct`, `union`, or `class` the resulting value is increased by 1 to reflect the extra word added to the argument list by the compiler to satisfy the Cray Research calling convention for functions returning an aggregate.
`long _dshiftl(`*x,y,n*`)`	The `_dshiftl` function returns a value generated by a left double-shift of the leftmost *n* bits of *y* into the rightmost *n* bits of *x*. If both the *x* and *y* arguments specify the same word in memory , a rotated shift occurs. The value of *n* must be in the range 0 through 64.

Function	Description
`long _dshiftr(`*x*,*y*,*n*`)`	The `_dshiftr` function returns a value generated by a right double-shift of the rightmost *n* bits of *x* into the leftmost *n* bits of *y*. If both *x* and *y* arguments specify the same word in memory, a rotated shift occurs. The value of *n* must be in the range 0 through 64.
`long _gbit(`*x*,*n*`)`	The `_gbit` function returns the value of the *n*th bit (from the right) of data word *x*. The value of *n* must be in the range 0 through 63.
`long _gbits(`*x*,*n*,*m*`)`	The `_gbits` function returns a value consisting of *n* bits extracted from *x*, starting at bit position *m* (from the right); the returned bits are right-justified. The value of *m* must be in the range 0 through 63.
`int _leadz(`*x*`)`	The `_leadz` function returns the number of leading 0's (from the left) in data word *x*.
`long _mask(`*n*`)`	The `_mask` function forms a mask of bits set to 1, left-justified or right-justified. If $0 \leq n \leq 63$, a left-justified mask of bits set to 1, *n* bits long, is formed. Otherwise, a right-justified mask of bits set to 1, $(128 - n)$ bits long, is formed. Value *n* must be in the range 0 through 128.
`long _maskl(`*n*`)`	The `_maskl` function returns a left-justified mask of *n* bits set to 1. Value *n* must be in the range 0 through 64.
`long _maskr(`*n*`)`	The `_maskr` function returns a right-justified mask of *n* bits set to 1. Value *n* must be in the range 0 through 64.

Function	Description
`int _numargs()`	The `_numargs` function returns the number of Cray words in the argument list passed to the function in which the call appears. The return value is undefined if `_numargs` is called from within an inline function. For this reason, this intrinsic function should never be called from within a file compiled with an automatic inlining level greater than 0. (See `-h inline`*n* in subsection 2.3.27.26, page 31, and the `_argcount` intrinsic function for more information.) When this intrinsic function is called from within a function that returns a `struct`, `union`, or `class` the result value is increased by 1 to reflect the extra argument added by the compiler to satisfy the Cray Research calling convention for functions returning an aggregate.
`long _pbit(`*x*`,`*n*`,`*y*`)`	The `_pbit` function returns a value consisting of *x* with its *n*th bit (from the right) set to the rightmost bit of data word *y*. The value of *m* must be in the range 0 through 63.
`long _pbits(`*x*`,`*n*`,`*m*`,`*y*`)`	The `_pbits` function returns a value consisting of *x* with *n* bits changed, starting at bit *m* (from the right), to equal the rightmost *n* bits of *y*. The value of *m* must be in the range 0 through 63.
`int _popcnt(`*x*`)`	The `_popcnt` function returns the number of bits set to 1 (the population count) in data word *x*.
`int _poppar(`*x*`)`	The `_poppar` function returns a 0 if the data word *x* has an even number of bits that are set to 1 and returns a 1 if the data word *x* has an odd number of bits that are set to 1. This is the parity of the population count for data word *x*.

Function	Description
`double _ranf()`	The `_ranf` function returns a random floating-point number in the range of $0.0 < x < 1.0$.
`long _rtc()`	The `_rtc` function returns the value of the real-time clock as a `long` value. Note that converting the value returned to `float` or `double` can result in loss of precision.

7.2 Intrinsics for Cray PVP systems

The following intrinsic functions are available on Cray PVP systems.

Function	Description
`void _cmr()`	The `_cmr` function assures completion of all memory references within the current CPU.
`void _EX(`*n*`)`	The `_EX` function performs a normal exit to the address in the exit address register (`EA`*n*) where *n* is an integer constant in the range 0 through 4. If *n* is in the range of 5 through 7, the function performs an exit to the exchange address register (`XA`). On all Cray Research systems except CRAY T90 systems, `_EX` does not take an argument; *n* is assumed to be 0.
`long _getvm()`	The `_getvm` function returns the contents of the vector mask register.
`void _mclr( )`	The `_mclr` function is available only on systems with bit matrix multiply (BMM) hardware. See subsection 7.5, page 140, for more information.
`long _mld(`x_2`)`	The `_mld` function is available only on systems with bit matrix multiply (BMM) hardware. See subsection 7.5, page 140, for more information.

Function	Description
`long _mldmx(`x_1`,`x_2`)`	The `_mldmx` function is available only on systems with bit matrix multiply (BMM) hardware. See subsection 7.5, page 140, for more information.
`long _mmx(`x_1`)`	The `_mmx` function is available only on systems with bit matrix multiply (BMM) hardware. See subsection 7.5, page 140, for more information.
`long _mul( )`	The `_mul` function is available only on systems with bit matrix multiply (BMM) hardware. See subsection 7.5, page 140, for more information.
`int _readSB(`*n*`)`	The `_readSB` function returns the value of shared-B register *n*. Value *n* must be in the range 0 through 7. *n* must be a constant.
`long _readSR(0)`	The `_readSR` function returns the value of status register 0. The only argument allowed is the constant value 0.
`long _readST(`*n*`)`	The `_readST` function returns the value of shared-T register *n*. Value *n* must be in the range 0 through 7. *n* must be a constant.
`void _semclr(`*n*`)`	The `_semclr` function clears semaphore *n*. Value *n* must be in the range 0 through 31. *n* must be a constant.
`long _semget( )`	The `_semget` function returns the value of the entire semaphore mask.
`void _semput(`*x*`)`	The `_semput` function stores the bit pattern in data word *x* into the semaphore mask (SM) register .
`void _semset(`*n*`)`	The `_semset` function sets semaphore *n*. Value *n* must be in the range 0 through 31. *n* must be a constant.

Function	Description
`void _semts(`*n*`)`	The `_semts` function performs a test-and-set operation on the semaphore *n*. Value *n* must be in the range 0 through 31. *n* must be a constant.
`void _setvm(`*x*`)`	The `_setvm` function sets the vector mask register to the value in data word *x*.
`void _writeSB(`*n*,*x*`)`	The `_writeSB` function writes the value of data word *x* to shared-B register *n*. Value *n* must be in the range 0 through 7. *n* must be a constant.
`void _writeST(`*n*,*x*`)`	The `_writeST` function writes the value of data word *x* to shared-T register *n*. Value *n* must be in the range 0 through 7. *n* must be a constant.

7.3 Intrinsics for Cray MPP systems

The following intrinsic functions are available on Cray MPP systems:

Function	Description
`void _memory_barrier( )`	The `_memory_barrier` function guarantees that all subsequent loads or stores will not access memory until after all previous loads and stores finish accessing memory.
`int _my_pe( )`	The `_my_pe` function returns the number of the processing element (PE) on which it is executing. The return value is an integer between 0 and `_num_pes()`– 1.
`int _num_pes( )`	The `_num_pes` function returns the number of PEs (and thus the number of tasks) available to your program. The function is evaluated at run time and cannot be used in constant expressions.

Function	Description

`void _remote_write_barrier( );`

The `_remote_write_barrier` function blocks the processor until all outstanding remote write request packets have been processed.

`void _write_memory_barrier( );`

The `_write_memory_barrier` function guarantees that all preceding store instructions have completed modifying memory before any subsequent loads or stores reference memory .

Intrinsics for Cray Research systems with IEEE floating-point hardware

7.4

The IEEE floating-point intrinsic functions access the IEEE floating-point hardware instructions. IEEE floating-point hardware is available on Cray MPP systems and on some CRAY T90 systems.

The following intrinsic function is available only on Cray Research systems with IEEE floating-point hardware:

Function	Description

`unsigned long _int_mult_upper(`*x*`,`*y*`)`

The `_int_mult_upper` function multiplies *x* and *y* as unsigned 64-bit integers and returns the upper 64 bits of the 128 bit unsigned result.

Intrinsics for bit matrix multiply hardware

7.5

The bit matrix multiply functions access the bit matrix multiply (BMM) hardware instructions. BMM hardware is not available on Cray MPP systems and some CRAY Y-MP systems. All BMM functions except `_mmx` and `_mclr` must appear in vectorizable short loops. The BMM instructions perform a matrix multiply of two bit matrices, x_1 and x_2, to provide result *y*.

Operands of the BMM functions reside in arrays; they are left-justified, zero-filled, and up to 64 bits in width. Operations on square bit matrices in which the number of elements is an integral of 64, can be performed using the `_mldmx` instruction to do both the load and multiply. The `_mld` function allows loading the x_2 argument separately for operations on nonsquare matrices of more than 64 elements. Arguments to the bit matrix multiply functions must be of type `long`.

Example:

```
for (i=0; i<=64; i++)
{
   matb[i] = _mld(b[i]);
   c[i] = _mmx(a[i]);
}
```

Or:

```
for (i=0; i<=64; i++)
{
   c[i] = _mldmx(a[i], b[i]);
}
```

In the preceding examples, the `b` matrix is loaded into the function unit where it is stored in its transposed form. That is, the rows and columns are exchanged: column 1 becomes row 1, column 2 becomes row 2, and so on. Transposing the input operand `b` allows the result matrix `c` to be formed as the mod-2 inner product of rows `a` with rows of `b`. This is done by ANDing the corresponding elements and using the result of an EXCLUSIVE OR on all the resulting bits.

Following is a summary of the BMM functions:

Function	Description
`long _mld(`x_2`)`	Loads the bit matrix function unit with a transposed matrix vector
`long _mmx(`x_1`)`	Performs the bit matrix multiply of x_1 times x_2 transposed
`long _mldmx(`x_1, x_2`)`	Combines the load and multiply functions

Function	Description
`long _mul( )`	Unloads the bit matrix function unit
`void _mclr( )`	Clears `BML` bit of the CPU status register and exchange package

Section 7

`_mld`: load transposed matrix

7.5.1

The `_mld` intrinsic function loads the bit matrix function unit with a matrix vector in transposed form. The function must appear in an unconditional vector loop and be detectable by the compiler as a short loop. If this intrinsic function appears in a conditionally vectorized loop or a nonvectorized loop, a fatal error will be issued. The `shortloop` directive may be used to indicate a safe condition.

The `_mld` intrinsic function returns a pseudo result, which should not be used because its contents are undetermined. If multiple references to `_mld` are made within a single loop, different names must be used to receive the pseudo result. This requirement forces optimization to consider earlier calls to `_mld` because the array that appears to receive the pseudo result seems to have no uses until it is redefined.

`_mmx`: matrix multiply

7.5.2

The `_mmx` intrinsic function performs the bit matrix multiply of x_1 times x_2 transposed. The argument to `_mmx` is the x_1 argument. The matrix contained in the function unit from a previous `_mld`, `_mmx`, or `_mldmx` invocation is the x_2 matrix that is used with x_1 to form the result. Intrinsic `_mmx` may appear in a scalar context and, in that case, may have a scalar argument.

`_mldmx`: load and multiply

7.5.3

The `_mldmx` intrinsic function combines the load and multiply functions. The second argument, x_2, is loaded into the bit matrix function unit prior to the multiply. Matrix x_2 is then multiplied by matrix x_1. This intrinsic must appear in a vector loop, as shown in the following example:

```
for (i=0; i<64; i++)
{
   dummy[i] = _mld(b[i]);
}
for (i=0; i<=512; i++)
{
   c[i] = _mldmx(a[i], b[i]);
}
```

`_mul`: unload bit matrix unit

7.5.4

The `_mul` intrinsic function unloads the bit matrix function unit. It returns the transposed value of the previously loaded x_2 vector. This function is performed by multiplying the unit's contents by an identity matrix. This intrinsic must appear in a vector short loop.

`_mclr`: clear BML bit

7.5.5

The `_mclr` intrinsic function clears the bit matrix loaded (BML) bit of the CPU status register and exchange package. You can reduce system overhead while the bit matrix portions of your code are inactive by declaring the bit matrix multiply (BMM) functional unit memory empty. This permits the operating system to omit the save and restore steps when switching the CPUs from user to user.

Conventions

7.5.6

Because the bit-matrix multiply (BMM) unit has a memory, you should be aware of the conventions needed to manage it. The UNICOS operating system maintains the bit matrix memory when disconnecting and connecting users from CPUs. Within your applications, the data, once loaded, remains until you alter it or use the _mclr intrinsic. Entry and exit sequences do not save and restore the contents of the unit. Recursive usage is not supported; you must manage such applications by unloading and saving any x_2 matrices that must be maintained.

Table of intrinsic functions

7.6

Table 4 provides a summary of all intrinsic functions discussed in this section in alphabetical order and indicates their availability on Cray Research computer systems. See the individual descriptions in this section for further restrictions.

Table 4. Summary of intrinsics

Function	Cray PVP systems	Cray MPP systems
`_argcount`	X	X
`_cmr`	X	
`_dshiftl`	X	X
`_dshiftr`	X	X
`_EX`	X	
`_gbit`	X	X
`_gbits`	X	X
`_getvm`	X	
`_int_mult_upper`	X†	X
`_leadz`	X	X
`_mask`	X	X
`_maskl`	X	X
`_maskr`	X	X

† Available only on Cray Research systems with IEEE floating-point hardware

Table 4. Summary of intrinsics (continued)

Function	Cray PVP systems	Cray MPP systems
`_mclr`	X††	
`_mld`	X††	
`_memory_barrier`		X
`_mldmx`	X††	
`_mmx`	X††	
`_mul`	X††	
`_my_pe`		X
`_numargs`	X	X
`_num_pes`		X
`_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	

†† This function is not available on some CRAY Y-MP systems

Table 4. Summary of intrinsics (continued)

Function	Cray PVP systems	Cray MPP systems
`_semset`	X	
`_semts`	X	
`_setvm`	X	
`_write_memory_barrier`		X†††
`_writeSB`	X	
`_writeST`	X	

††† Available only on CRAY T3E systems.

Debugging C/C++ Code [8]

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 subsection 2.3.7, page 11.
- The `-h trunc` and `-h notrunc` options help identify numerically unstable algorithms. For more information, see subsection 2.3.27.55, page 47.
- The `-h bounds` option and the `#pragma _CRI [no]bounds` directive (`cc`(1) only) let you check pointer and array references. The `-h bounds` option is described in subsection 2.3.27.9, page 25. The `#pragma _CRI [no]bounds` directive is described in subsection 4.3.1, page 77.
- The `#pragma _CRI message` directive lets you add message text to sections of code where you suspect problems. The `#pragma _CRI message` directive is described in subsection 4.3.3, page 81.
- The `#pragma _CRI opt` and `#pragma _CRI noopt` directives let 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 subsection 4.3.4, page 82.

Cray TotalView debugger

8.1

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*, publication IN–2502.

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.

Compiler debugging options

8.2

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, return statements, and labels.

- `-Gp`

  If you specify the `-Gp` debugging option, the Cray TotalView debugger allows you to set breakpoints at function entry, return statements, and labels, and at the first executable statement in a block (unless the block is empty).

- `-Gn`

  If you specify the `-Gn` debugging option, the Cray TotalView debugger allows you to set breakpoints at any executable statement.

- `-g`

  The `-g` option is similar to `-Gn` in that you can set breakpoints at any executable statement. However, unlike `-Gn`, the `-g` option forces all compiler optimizations to be disabled. In general, the `-g` option is equivalent to `-Gn -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. The `-G` option does not directly affect optimization. Thus, the optimization analysis at `-O3` is the same with `-Gn` as it is with `-Gf`. However, because `-Gf` generates fewer breakpoints than `-Gp`, optimizations attempted at `-Gf` are more likely to succeed, and optimizations attempted at `-Gp` are more likely to succeed than those at `-Gn`.

In addition, 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.
- Setting a breakpoint at the first statement in a vectorized loop allows you to stop and display at each vector iteration when the `-Gp` option is specified. However, setting a breakpoint at the first statement in an unrolled loop may not allow you to stop at each vector iteration.

Note: While it generates the same number of breakpoints as `-Gn`, the `-g` option inhibits optimization as though `-O0` was specified.

8.3 Using the Cray TotalView debugger on Cray MPP systems

One of the basic applications of Cray TotalView is to use the core file of an aborted program to locate the statement that caused the abort. To allow attachment of the process within Cray TotalView, compile your program using the -Gn option and run the executable using the -debug option. If the program aborts, a dump of the failing processing element (PE) is written to the mppcore file. You can then run Cray TotalView to examine information about the job. For example:

```
cc -T cray-t3d -X 128 -Gn pgm.c
./a.out -debug &           # Creates core file
totalview mppcore
```

Another , more expensive, way to analyze an aborted program is to run it using the -freeze option. This causes Cray TotalView to display the stack trace (calling sequence), memory contents, and other information about the job's state at the abort. For example:

```
cc -T cray-t3d -X 128 pgm.c
./a.out -freeze &
totalview
```

To debug your program separately from an abort, compile it by specifying the -Gn option. Then invoke the executable program as an argument to TotalView. For example:

```
cc -Gn pgm.c
totalview ./a.out
```

Interlanguage Communication [9]

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*, publication SN–3009. The calling sequence is described in detail on the `callseq`(3) man page, which is included in the *Application Programmer's Library Reference Manual*, publication SR–2165.

Interlanguage communication with Cray Standard C

9.1

Cray Standard C provides a mechanism for declaring external functions that are written in other languages. This allows the C programmer to write portions of an application in 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.

This section describes how to call assembly language and Fortran programs from a C program. It also discusses the issues related to calling C programs from other languages.

Section 9

Calling assembly language functions from a C program

9.1.1

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 used by 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 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*, publication SR–2403.

Cray Assembly Language (CAL) functions on Cray PVP systems

9.1.1.1

The use of Cray Assembly Language (CAL) on Cray PVP systems is described in the *Cray Assembly Language (CAL) for Cray PVP Systems Reference Manual*, publication SR–3108.

On Cray PVP 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.

Cray Assembler for MPP (CAM) functions on Cray MPP systems

9.1.1.2

The use of the Cray Assembler for MPP (CAM) on Cray MPP systems is described in the *Cray Assembler for MPP (CAM) Reference Manual*, publication SR–2510.

On Cray MPP 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.

Calling Fortran routines from a C program

9.1.2

This subsection describes aspects of calling between C functions and Fortran: requirements and guidelines, MPP considerations, argument passing, array storage, logical and character data, and accessing blank common from C programs.

Requirements

9.1.2.1

Keep the following points in mind when calling Fortran functions from C:

- Fortran uses the call-by-address convention, and C uses the call-by-value convention. See subsection 9.1.2.3, page 153.
- Fortran arrays are column major, and C arrays are row major. This indicates which dimension is indicated by the first value in an array element subscript. See subsection 9.1.2.4, page 154.
- Fortran character pointers and C character pointers are incompatible. See subsection 9.1.2.6, page 156.

- Fortran logical values and C Boolean values are not fully compatible. See subsection 9.1.2.6, page 156.
- Fortran character variables are not terminated with a `NULL` character and C character variables must be terminated with a `NULL` character. When copying Fortran character variables to C, the programmer must provide the `NULL` termination.
- Fortran character variables are padded with blanks to the end of the variable.

MPP considerations
9.1.2.2

Cray MPP systems involve the following additional considerations:

- The C language `float` type does not match the Fortran `REAL` type. The `float` type is 32 bits for Cray Standard C on MPP systems; the Fortran `REAL` type is 64 bits. However, Fortran includes the `REAL*4` type that matches the C language `float` type.
- A Fortran character descriptor (`_fcd`) consists of two 64-bit words on Cray MPP systems.

Argument passing
9.1.2.3

Because Fortran subroutines expect arguments to be passed by pointers rather than by value, 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 on the understanding that the function can change the value of the pointer's object. (All array arguments are passed by this method.) This capability is analogous to the Fortran method of passing arguments.

Array storage
9.1.2.4

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 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 and Fortran, its dimensions are declared and referenced in C in the opposite order in which they are declared and referenced in Fortran. Since arrays are zero-based in C and one-based in Fortran, in 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 would be:

```
int a[2][3];
```

Section 9

The following table shows how to access elements of the array from Fortran and C:

Fortran	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]

Other guidelines 9.1.2.5

The following guidelines can help you communicate between C and Fortran (the burden of communication rests with the C program):

- The name of a called Fortran function must consist of 31 or fewer characters; all of the characters must be uppercase letters or digits.
- External C variables are stored in common blocks of the same name, making them readily accessible from Fortran programs if the C variable is uppercase.
- 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. Because Cray Standard C is row major, arrays of more than one dimension require special care.
- When a C function invokes a Fortran subprogram, all actual parameters of the C function must be addresses (array names are considered addresses).
- When a Fortran subprogram invokes a C function, all formal parameters of the C function must be declared as pointers.

Logical and character data 9.1.2.6

Logical and character data need special treatment for calls between C and Fortran. Fortran has a character descriptor that is incompatible with a character pointer in C. The techniques used to represent logical (Boolean) values also differ between 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 character pointer to a Fortran character descriptor.
`_fcdtocp`	Conversion utility that converts a Fortran character descriptor to a 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 function `_cptofcd`(3) in the *UNICOS System Libraries Reference Manual*, publication SR–2080. Using type `_fcd` and these conversion utilities, passing character data between Fortran and C is straightforward.

Section 9

9.1.2.7 Accessing blank common from 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 `COMMOM` or `TASK COMMON` 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 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 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 source code:

```
#include <stdio.h>

struct st
{
  float a;
  float b[10];
} *ST;

fortran struct st *MYCOMMON(void);
fortran void FCTN(void);

main()
{
    int i;

    ST = MYCOMMON();
    ST->a = 1.0;
    for (i=0; i<10; i++)
        ST->b[i] = i+2;
    printf("\n\n In 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 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 source code is as follows:

```
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:

```
STA = 1.
STB = 2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.,  10.,  11.
```

Section 9

C and Fortran example 9.1.2.8

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
```

Section 9

The previous C function and Fortran subprogram are executed by the following commands and produce the following output:

```
$ cc -c main.c
$ cf77 -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
$
```


Calling a C function from an assembly language or Fortran program

9.1.3

A C function can be called from Fortran or assembly language. When calling from Fortran, keep in mind the information in subsection 9.1.2, page 152.

The examples that follow illustrate 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.

The following example illustrates a Fortran program that calls a C function:

```
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
```

Section 9

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
$ cf77 -c main.f
$ segldr 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
$
```

The following example demonstrates how external C variables are accessible in Fortran 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`:

```
struct
{
   int i;
   double a[10];
   long double d;
} ST;

main()
{
   int i;

   /* initialize struct ST */
   ST.i = 12345;

   for (i=0; i<0; 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<0; 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
$ cf77 -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
$
```

Other types of external C variables can be accessed from Fortran common blocks. The previous examples in this section demonstrate how `float` types can be accessed using this method.

Interlanguage communication with Cray C++ 2.0

9.2

Cray C++ provides a mechanism for declaring external functions that are written in other languages. This allows the C++ programmer to write portions of an application in 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++.

This section describes how to call C and Fortran 90 programs from a C++ program. It also discusses the issues related to calling C++ programs from other languages.

Calling a C program from a C++ program

9.2.1

The `extern C` keyword is required when declaring an external function that is written in C. Normally the C++ compiler will mangle names to encode information about the function's prototype in the external name. This prevents direct access to these names from a C program. The `extern C` keyword will prevent the C++ compiler from performing name mangling. The `extern C` keyword can be applied to any external declaration, such as a function or data declaration.

The following requirements must be considered when calling a function written in C from C++:

- The C++ program that calls the C routine must declare the C routine as `extern C`. This inhibits normal name mangling of the external name.
- The main program must be a C++ program.

C++ also allows data structures to be shared between C++ functions and C functions. There are some C++ data structures that are not accessible to C programs (such as classes). The following data structures can be shared directly:

- Scalar data items (`char`, `short`, `int`, `long`, `float`, `double`, `long double`)
- 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 of scalar data items, structures, and unions that are declared identically in C and C++.
- Pointers to scalar data items, structures, and unions that are declared identically in C and C++.

External data shared between C++ and C programs must include the `extern C` keyword in the C++ declaration.

In the following example, a C function (`C_add_func`) is called by the C++ main program:

```
        C++ Main Program

#include <iostream.h>

// The following style declares a function as
// and a global data item as extern ”C”.

extern ”C” {
int C_add_func(int, int);
int global_int = 123;
};

void main() {
  int res, i;

  cout << ”Start C++ Main Program” << endl << endl;

  // -----------------------------------------------------
  // Call C routine 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 << endl;

  cout << ”End C++ Main Program” << endl;

}
```

The C function (`C_add_func`), called from the C++ main program in the preceding example, is as follows:

```
#include <stdio.h>

extern int global_int;

int C_add_func(int p1, int p2) {
    printf("\tStart C routine C_add_func.\n");
    printf("\t\tParam1     = %d\n",p1);
    printf("\t\tParam2     = %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 examples is as follows:

```
Start C++ Main Program

Call C C_add_func
        Start C routine C_add_func.
                Param1     = 10
                Param2     = 20
                Global Int = 123
Result of C_add_func = 30
End C++ Main Program
```


9.2.2 Calling a Fortran program from a C++ program

In addition to calling C programs from C++, it is possible to call Fortran programs from C++. The rules for calling Fortran from C++ are as follows:

- The `extern C` keyword is required on the declaration of the Fortran routines in the C++ file. The name of the routine in the C++ declaration must be all uppercase.

- The `extern C` keyword is required on data definitions in the C++ file that need to be accessed by Fortran. The name of the external data in the C++ declaration must be all uppercase.
- The main program must be in C++.
- The `fortran` keyword cannot be used because it does not exist in C++. For more information, see subsection 10.2, page 186.

In addition, the following rules for calling Fortran from C also apply to calling Fortran from C++:

- Fortran uses the call-by-address convention, and C uses the call-by-value convention. This means only pointers or addresses should be passed to Fortran routines.
- Fortran arrays are column major, and C arrays are row major. This indicates which dimension is indicated by the first value in the array subscript.
- Fortran character pointers and C character pointers are not compatible.
- Fortran logical values and C Boolean values are not fully compatible.

The following example illustrates how a Fortran 90 program can be called from a C++ program:

```
#include <iostream.h>
#include <fortran.h>
// The following style declares one function as extern ”C”.

extern ”C” int FORTRAN_ADD_INTS (int *arg1, int &arg2);

void main() {
    int num1, num2, res;
    cout << ”Start C++ Main Program” << endl << endl;

// Call FORTRAN routine to add two integers and return result.
// Note that the second argument is a reference parameter so
// it is note 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 Program” << endl;
}
```


The Fortran 90 program that is called from the C++ main program 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 calling sequence illustrated in the preceding examples is as follows:

```
Start C++ Main Program

Before Call to FORTRAN_ADD_INTS
  FORTRAN_ADD_INTS, Arg1,Arg2 =  10,  20
Result of FORTRAN Add = 30

End C++ Main Program
```

Calling a Cray Assembly Language (CAL) program from C++
9.2.3

The requirements for calling a Cray Assembly Language (CAL) program from C++ are the same as the requirements for calling a CAL program from C, with the addition of the `extern C` keyword on the declaration.

Calling a C++ program from a C program
9.2.4

The requirements for calling a C++ program from a C program are as follows:

- The routines called from C must have the `extern C` keyword.
- The main program must be a C++ program.

Calling a C++ program from a Fortran 90 program
9.2.5

The requirements for calling a C++ program from a Fortran 90 program are as follows:

- The C++ routine must have the `extern C` keyword.
- The main function must be in C++.
- The requirements listed in subsection 9.1.3, page 162, for calling C programs from Fortran also apply.

Shared data on Cray MPP systems

9.3

Shared data on Cray MPP systems is data that is physically distributed across the memories associated with processing elements (PEs). For a complete description of shared data, see the *Cray MPP Fortran Reference Manual,* publication SR–2504.

However , when *shared-to-private coercion* is performed on shared array arguments passed to a C function from a Fortran routine, the C function can access those elements that are local to the PE. You must ensure that only the local portion of the array is referenced.

The `_S2PC` library function performs shared-to-private coercion, if necessary . The format used to call the `_S2PC` library function is as follows:

```
extern void *_S2PC(void *ptr_to_shared ,
     size_t element_size );
```

The `_S2PC` function accepts a pointer argument (*ptr_to_shared*) and the size of each element in the array (*element_size*), and returns a pointer to the corresponding local element. For example, if an array with 16 elements is distributed across four PEs, the distribution might be as shown in Figure 2.

PE0	PE1	PE2	PE3
A(1)	A(5)	A(9)	A(13)
A(2)	A(6)	A(10)	A(14)
A(3)	A(7)	A(11)	A(15)
A(4)	A(8)	A(12)	A(16)

Figure 2. Block distribution, one dimension

Section 9

In the example shown in Figure 2, each PE has four elements. The first four elements are on PE0, the next four elements are on PE1, the next four elements are on PE2, and the last four elements are on PE3. If a pointer to this kind of array is passed to a C function, the C function can use the _S2PC library function to gain access to the local elements. That is, PE0 can access elements 1 through 4, PE1 can access elements 5 through 8, PE2 can access elements 9 through 12, and PE3 can access elements 13 through 16.

If a pointer to private data is passed to the _S2PC function, it is returned with no modification. Therefore the _S2PC function can be safely called with either a pointer to shared data or a pointer to private data.

The following example demonstrates how each PE can reference local elements by using the _S2PC library function. The Fortran program passes a shared argument to the C function, which prints the local elements.

```
      PROGRAM FS2PC
      INTEGER L, N
      PARAMETER(N=64)     ! Total number of elements
      PARAMETER(L=N/N$PES) ! Number of local elements
      COMMON /SHARED_DATA/ X
      REAL X(N)
CDIR$ SHARED X(:BLOCK)

C If N=8 and N$PES=2 then after INIT is called, IA
C  will look like:

C    IA  = 1 2 3 4 5 6 7 8
C          ------- -------
C            pe0     pe1

C Function PRINT_LOCAL prints the elements local to each PE

      CALL INIT(IX, L)        ! Initialize IA
      CALL PRINT_LOCAL(IX, L)  ! Print local elements
      END
```

(continued)

Section 9

```
       SUBROUTINE INIT(IY, J)
       INTEGER IY(J)
       INTRINSIC MY_PE   ! Identifies the PE number
       DO I = 1, J
          IY(I) = MY_PE() * J + I
       ENDDO
       RETURN
       END  #include <stdio.h>

       #include <fortran.h>

       void PRINT_LOCAL(void *p, int *len) {
         int *local;
         int  buf_len;
         int  i;
         int  n;
         char buffer[1024];

         local  = _S2PC(p, sizeof(int));
     /* Point to local elements */

     /* Buffer the output so only one I/O operation is performed. */
     /* This is needed to ensure that each PE does atomic I/O. */
     /*  The %n tells how many characters are written to buffer. *

        sprintf(buffer, "PE %d:  %n", _my_pe(), &n);
        for (i = 0; i < *len; i++) {
          sprintf(&buffer[n], "%d  %n", *local++,
 &buf_len);
          n += buf_len;
        }
        printf("%s\n", buffer);

      }
```

Section 9

The output from the previous example is as follows (the order can vary from run to run):

```
PE 0:  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16
PE 3:  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64
PE 1:  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32
PE 2:  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48
```

The next example is more complicated. It computes local sums on each PE and compares the result against an expected value.

```
      PROGRAM FPGM
      INTEGER L, N
      PARAMETER(N=64)      ! Total number of elements
      PARAMETER(L=N/N$PES) ! Number of local elements
      COMMON /MY_DATA/ IA
      INTEGER IA(N)
CDIR$ SHARED IA(:BLOCK)

      INTRINSIC MY_PE      ! Identifies the PE number

C If N=8 and N$PES=2 then after INIT is called, IA will
C look like:

C    IA  = 1 2 3 4 5 6 7 8
C          ------- -------
C            pe0     pe1

C The LCHK function double checks the local sum.  Each PE has
C a range of elements [[ L*MY_PE .. L*(MY_PE+1) ]].
C Therefore the local sum can also be computed as:

      LCHK(M) = M*M*MY_PE() + (M*M + M) / 2

C Function ISUM adds the local elements on each PE.

      CALL INIT(IA, L)        ! Initialize IA
      IRESULT = ISUM(IA, L)    ! Compute local sum
C Compare with LCHK:
      IF (IRESULT .EQ. LCHK(L)) THEN
        PRINT *, 'PE ', MY_PE(), ' Success'
      ELSE
        PRINT *, 'PE ', MY_PE(), ' Failure'
      ENDIF
      END
```

(continued)

Section 9

```
        SUBROUTINE INIT(IY, J)
       INTEGER IY(J)
       INTRINSIC MY_PE   ! Identifies the PE number
C
       DO I = 1, J
         IY(I) = MY_PE() * J + I
       ENDDO
       RETURN
       END

#include <stddef.h>
extern void *_S2PC(void *, size_t)

int ISUM(void *p, int *len) {
  int  lsum = 0;
  int *local;
  int  i;

  local  = _S2PC(p, sizeof(int));
/* point to local elements */
  for (i = 0; i < *len; i++) {
    lsum += *local++;   /* compute local sum */
  }

  return lsum;
}
```

The resulting output from the previous example for four PEs is as follows (The PE order can vary from one run to another):

```
PE 0 Success
PE 3 Success
PE 1 Success
PE 2 Success
```

Cray C and C++ Extensions [10]

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

Restricted pointers are an extension to both the C and C++ standards.

Extensions to only the C standard include the following (not available in C++):

- `fortran` keyword
- Complex data types
- Hexadecimal floating-point constants
- Variable length arrays

Restricted pointers

10.1

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 Cray PVP systems is that they can increase the number of loops that the compiler can vectorize. On Cray MPP 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 subsection 2.3.27.42, page 40).

Section 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 `-Drestrict=` option or by using the following code:

```
#ifdef _CRAYC
#define RESTRICT restrict
#else
#define RESTRICT
#endif
```

If the preceding header is 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 systems:

```
int * RESTRICT p;
```

On Cray PVP systems the compiler ignores uses of `restrict` that it currently cannot vectorize. Some cases that are ignored are discussed in subsection 10.1.7, page 186.

Function parameters
10.1.1

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++;
   }
}
```

Alternative syntax

10.1.2

A function parameter declared as `Type p[restrict]` is interpreted as though it were `Type * restrict p`.

This alternative syntax is an extension of the interpretation, required by the C standard, of a parameter declaration of the form `Type p[]` as though it were `Type *p`. This syntax is especially convenient for parameters that are pointers to variable length arrays, because it allows a more array-like syntax.

The following example illustrates the use of the alternative syntax:

```
void f(int n, double a[restrict][n]);
```

The code from the previous example can be used instead of the following:

```
void f(int n, double (* restrict a)[n]);
```

File scope

10.1.3

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]);
}
```

Block scope

10.1.4

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);
}
```

Unrestricted pointers

10.1.5

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.

Comparison with the `ivdep` pragma
10.1.6

Restricted pointers and the `ivdep` pragma 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.

Implementation limits
10.1.7

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.

`fortran` keyword
10.2

Note: The `fortran` keyword is not allowed in 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 subsection 9.1.2.8, page 159.

Complex data types

10.3

Note: Complex data types are available in Cray Standard C only. Complex data types are supported in Cray C++ 2.0 through the complex class library.

Cray Standard C provides `complex float` and `complex double` data types for use on all Cray Research 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 header `<complex.h>` 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.

Either of the two complex data types is represented in memory with two contiguous parts, the *real* part and the *imaginary* part, whose characteristics agree with those of the corresponding real types. For example, the imaginary part of a `complex float` 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 unavailable.

Section 10

Complex usage

10.3.1

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

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

```
complex double 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>
```

Conversion to and from complex

10.3.2

A binary operator with one complex operand and one real operand causes the real type to be promoted to a complex type before the operation is performed. When real is promoted to complex, 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 complex is demoted to real, the value of its imaginary part is discarded, and its real part is demoted according to the demotion rules for the corresponding floating type.

Cray Research does not currently support a `complex long double` type, so if one operand is `complex double` and the other is `long double`, this is treated as an error. One way to write portable code (if also writing for other computer systems that support `complex long double`), is as follows:

```
#if _CRAY
typedef complex double COMPLEX_LONG_DOUBLE
#else
typedef complex long double COMPLEX_LONG_DOUBLE
#endif
```

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.

Added math functions that take `complex double` arguments are listed below and described in the *UNICOS System Libraries Reference Manual*, publication SR–2080.

`csin`	`cexp`	`cpow`	`cabs`	`conj`
`ccos`	`clog`	`csqrt`	`cimag`	`creal`

Arithmetic conversions
10.3.3

Adding complex types to the C language affects the usual arithmetic conversions as specified in the standard. As a result, conversions are performed as follows:

1. If one operand is `complex float` or `complex double` and the other is `long double`, an error message is issued because type `complex long double` is not supported.
2. Else, if either operand is `complex double`, the other operand is converted to `complex double`, and the result is `complex double`.
3. Else, if either operand is `complex float`, the other operand is converted to `complex float`, and the result is `complex float`.

Section 10

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

5. Else, if either operand is `double`, both are converted to `double`, and the result is `double`.

6. Else, if either operand is `float`, both are converted to `float`, and the result is `float`.

7. Else, any operands of type `char`, `signed char`, `unsigned char`, `short`, or `unsigned short` are converted to `int`. If all values of a bit field can be represented as `int`, they are converted to type `int`; otherwise, they are converted to `unsigned int`. These are called *integral promotions*. Then:

   a. If either operand is `unsigned long`, both are converted to `unsigned long`, and the result is `unsigned long`.

   b. 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`.

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

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

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

Hexadecimal floating-point constants

10.4

Note: Hexadecimal floating-point constants are not available in Cray C++ 2.0.

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.

Section 10

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 Cray PVP 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
```

Section 10

Variable length arrays

10.5

> **Note:** Variable length arrays are not supported in Cray C++ 2.0.

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.

Declarator restrictions

10.5.1

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. Member objects of structure and union types are not allowed to have variable length array type. An object that has a type containing one or more derived declarator types of variable length array type is said to be *variably qualified*.

Variable length arrays can be used as function parameters. 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.

Variable length array (VLA) declarators

10.5.2

The size of a variable length array (VLA) dimension must be of integer type and cannot be 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 to specify the size of an array may produce side effects; 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.

Two array types are compatible when one or both are variable length arrays if they both have compatible elemental types and each specified dimension is constant and of equal value. Nonconstant dimension specifiers for one array must evaluate, at program execution time, to values equal to those of the corresponding constant or nonconstant dimension specifiers of the other array . Undefined behavior results if corresponding dimension specifiers 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;
func()
{
  int a[n][6][m];
  int (*p)[4][n];
  int c[n][n][6][m];
  int (*r)[n][n][n+1];

  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;
void main()
{
  int (*q)[dim4][8][dim6];
  int (*r)[dim6][8][1];

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

Function declarators and variable length arrays

10.5.3

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.

When ordinary identifiers (not enumeration constants) are used in a parameter type list to specify variable length array types, all such variable types are incomplete, and evaluation of the ordinary identifiers is suspended until the list is terminated with a definition. Thus, the order in which parameters are declared (and, accordingly, the order in which variable length array modifiers are specified) is not important because the type is not completed or in need of completion until a function definition is found.

The manner in which the declaration is handled depends on whether the function definition is found, as follows:

- If no function definition is found, the array modifier types affected do not need to be completed, and the associated dimension specifier expressions are not evaluated. 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[*][*]);
```

- If the function definition is found, ordinary identifiers (that are not enumeration constants), that are found in suspended size expressions are resolved according to visible declarations in both file and current function prototype scopes. All suspended expressions are evaluated (and thus, the incomplete variable length array types completed) simultaneously. An indirect or direct mutual reference (for example, using the `sizeof` operator) among incomplete variable length array types results in undefined behavior.

  In the following example, the prototype has a variably qualified parameter:

```
void addscalar(int n, int m, double a[n][n*m + 300], double x);
double A[4][308];

main()
{
  addscalar(4, 2, A, 3.14);
}

void addscalar(int n, int m, double a[n][n*m + 300], double x)
{
  int i,j,k=n*m+300;

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

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[\^][*]);
```

As a final example, the following function declarator used as a function definition would yield an error because the variable length types associated with each parameter are completed simultaneously and mutual reference makes the variably qualified parameter types incomplete:

```
int func (int (*p)[sizeof(*q)],
          int (*q)[sizeof(*p)]) {  }
```

Variable length array type definitions

10.5.4

Type definition declarations that specify a variable length array type must have block scope. The size of the variable length array 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.

Section 10

The following example shows a legally declared variable length array:

```
main()
{
  extern void func(int);
  func(20);
}

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

The following example shows a declared variable length array that is not valid:

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

The following example declares variable length arrays 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];
     }
}
```

`sizeof` operator and variable length arrays

10.5.5

When the `sizeof` operator is applied to an operand that has array type, the result is the total number of bytes in the array. For variable length array types, the result is not a constant expression and is computed at program execution time. For example:

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

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

The output from the preceding example is `80`.

`switch` statement
10.5.6

The controlling expression must not cause control to jump into a block where a variably qualified object has been declared. This error is not diagnosed by the compiler.

```
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;
   }
}
```

`goto` statements

10.5.7

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;
  }
```

`setjmp` and `longjmp` functions

10.5.8

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);
}
```

Libraries and Loaders [A]

This appendix describes the libraries that are available with the Cray C++ Programming Environment, 2.0 release and the UNICOS loaders, SEGLDR and MPPLDR.

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(1), CC(1), or c89(1) commands to invoke the compiler and the program compiles without errors, the loader is called. Including 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`, `mppld`, `segldr`, or `mppldr`), the `-h keep=`*files* option (see subsection 2.3.27.32, page 35) must be specified on the command line when compiling source files.

The UNICOS loaders, SEGLDR and MPPLDR, can be accessed through the following methods:

- You can access the loader directly by using the `segldr`(1) and `mppldr`(1) commands.
- You can use the `ld`(1) command of SEGLDR or the `mppld`(1) command of MPPLDR, which provides the traditional UNIX `ld` interface.
- You can leave the choice of loader to the `cc`, `CC`, or `c89` command.

Using the `cc`, `CC`, or `c89` command to call the loader 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.

You sometimes pay for these advantages, however, with slower loading or more memory use.

UNICOS loader for Cray PVP systems (SEGLDR)

A.2.1

The default loader on all Cray PVP systems is SEGLDR. The `CC`(1), `cc`(1), and `c89`(1) commands call SEGLDR by using the `ld`(1) command. Because SEGLDR was designed specifically for use with Cray language processors and Cray PVP 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`(1) command line or directives in a directives file. For more information, see the `segldr`(1) man page in the *UNICOS User Commands Reference Manual*, publication SR–2011, and the *Segment Loader (SEGLDR) and* `ld` *Reference Manual* , publication SR–0066.

UNICOS loader for Cray MPP systems (MPPLDR)

A.2.2

The default loader on all Cray MPP systems is MPPLDR. The `CC`(1), `cc`(1), and `c89`(1) commands call MPPLDR by using the `mppld`(1) command. Because MPPLDR was designed specifically for use with Cray language processors and Cray MPP systems, it offers several advantages. You can control MPPLDR operations with options on the `mppldr`(1) command line or directives in a directives file. For more information, see the `mppldr`(1) man page in the *UNICOS User Commands Reference Manual*, publication SR–201 1, and the *Cray MPP Loader User's Guide*, publication SG–2514.

Cray C++ Dialects [B]

Section B

This appendix details the features of the C++ language that are accepted by the Cray C++ compiler, including certain language dialects and anachronisms. Users should be aware of these details, especially users who are porting codes from other C++ environments or from Cray C++ 1.0.

C++ conformance

B.1

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. This is essentially the same language as defined by the language reference for `cfront` version 3.0.*x*, with the addition of exceptions.

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 unwittingly 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 `-h[no]anachronisms` and `-h[no]conform` command-line options are described in subsection 2.3.27, page 22.

The following features which are not in ARM but are in the *X3J16/WG21 Working Paper*, are also supported:

- Implicit conversion of `T**` to `const T *const *` and similiar conversions.
- New-style casts (for example, `const_cast`).
- The restriction on using a dependent statement of an `if`, `while`, `do-while`, or `for` loop is removed and is considered to be a scope.

- The expression that is tested in an `if`, `while`, `do-while`, or `for` loop as the first operand of a `?` operator or as an operand of the `&&`, `||` or `!` operators can have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.
- Use of a global-scope qualifier in member references of the form `x.::A::B` and `p->::A::B` is allowed.
- The precedence of the third operand of the `?` operator is changed.
- If control reaches the end of the `main()` routine, and `main()` has an integral return type, it is treated as if a `return 0` statement were executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a constructor. The temporary object created gets the same default initialization to 0 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.

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

- Virtual functions in derived classes cannot return a type that is the derived-class version of the type returned by the overridden function in the base class.
- The digraph and macro forms (for example, `<%,bitand`) for certain operators and punctuators are not accepted.
- `wchar_t` is not treated as a keyword.
- It is not possible to overload operators using functions that take `enum` types and no class types.
- Definition of nested classes outside of the enclosing class is not allowed.

- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are not yet implemented.
- Classes are not assumed to always have constructors, and the distinction between trivial and nontrivial constructors is not implemented.
- The `mutable` keyword is not implemented.
- Name spaces are not implemented.
- `enum` types cannot contain values larger than can be contained in an `int` type.
- Type qualifiers are not retained on `rvalues`, in particular, on function return values.
- Type `bool` is not implemented.
- Explicit qualification of template functions is not implemented.
- Explicit instantiation of templates in the style of N0274/93–0067 is not implemented.
- Name binding in templates in the style of N0288/93–0081 is not implemented.
- The scope of a variable declared in a `for` loop is the whole surrounding scope, not just the loop.
- Member templates are not implemented.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is not evaluated. This is as required by the ARM. The working paper , however, requires that `f()` be evaluated.
- `(p->*pm) = 0` cannot yet be written as `p->*pm = 0` (the syntax still matches the ARM and cfront).
- `typename` in templates is not implemented.
- Non-converting constructors are not implemented.
- Class name injection is not implemented.

- Overloading of function templates (partial specialization) is not implemented.
- Partial specialization of class templates is not implemented.
- Placement delete is not implemented.
- Putting a `try`/`catch` around the initializers and body of a constructor is not implemented.
- The notation `:: template` (and `->template`, and so forth) is not implemented.

Section B

Anachronisms accepted

B.2

Anachronisms are enabled by using the `-h anachronisms` command-line option (see subsection 2.3.27.4, page 23). When anachronisms are enabled, the following anachronisms are accepted:

- Overloading 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 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.

Extensions accepted in normal C++ mode

B.3

The following extensions are accepted in all C++ modes:

- A `friend` declaration for a class can omit the `class` keyword as shown in the following example:

```
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();
}
```

- `operator()()` functions can have default argument expressions. A warning message is issued.

- In a reference to a class template, the argument corresponding to a non-type parameter can require a promotion or standard conversion. ARM requires an exact match, as shown in the following example:

```
template<float F> class S {};
S<2.0> x;
```

- In a call to be resolved from a function template, the following extensions apply:
  - Trivial conversions involving references, array and function type decay to pointer, and addition of type qualifiers is allowed, as shown in the following example:

```
template <class T> void f(const T &p) {}
void m() {int i; f(i);}
```

  - Parameters that do not involve template types can require trivial conversions and/or a cast to a base class, as shown in the following example:

```
struct A {};
struct B : public A {} *p;
template<class T> void f(T, A*);
void m() {f(1, p);}
```

- A pointer to a constant type can be deleted.

The following extensions are accepted, but are flagged if the `-h conform` command-line option is specified:

- A translation unit (input file) cannot contain 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. A warning is issued in strict ANSI mode (`-h conform`), or a remark is issued.
- 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.) A warning is issued.
- An empty declaration (a semicolon preceded by nothing) is allowed. A remark is issued.
- An initializer expression that is a single value and is used to initialize an entire static array, struct, or union need not 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.
- 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. A warning message is issued.
- Benign redeclarations of `typedef` names are allowed. That is, a `typedef` name can be redeclared in the same scope with the same type. A warning message is issued.
- 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.

Section B

- Assignment and pointer difference 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. A warning is issued.
- 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).
- A pointer to `void` can be implicitly converted to or from a pointer to a function type. A warning is issued.
- Adjacent wide and normal string literals are not concatenated.
- In character and string escapes, if the character following the ending quotation mark (`"`) has no special meaning, the value of the escape is the character itself. For example, `""s" == "s"` issues a warning.
- Restricted pointers are allowed. For more information, see subsection 10.1, page 181.

Extensions accepted in cfront compatibility mode

B.4

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 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.

- No warning is issued when an `operator()()` function has default argument expressions.

- 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
```

- A reference to a pointer type can 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
```

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

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

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

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

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

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

Compiler Messages [C]

Section C

This appendix describes how to use the UNICOS 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 subsection.

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 UNICOS message system, see the *UNICOS Message System Programmer's Guide*, publication SG–2121, or the message system section of the *UNICOS System Libraries Reference Manual*, publication SR–2080. The introduction to that section can be viewed online as the `message`(3) man page.

Expanding messages with the `explain` command

C.1

If you have interactive use of a Cray Research system, 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 is `cc` (lowercase). The prefix for Cray C++ is `CC` (uppercase).

In the following sample dialog, the `cc`(1) command invokes the compiler on source file `bug.c`. Message `cc-480` is displayed. The `explain`(1) command displays the expanded explanation for this message.

```
> cc bug.c
cc-480 cc: ERROR File = bug.c, Line = 6
  Braces are missing from the aggregate initializer "0.".

TOTAL ERRORS DETECTED IN bug.c: 1
> explain cc-480

CC-ERROR:  Braces are missing from the aggregate initializer "init".

Array, structure, and union initializers are specified in initializer
lists.  An initializer list is a comma-separated list of initializers;
initializer lists are required to be enclosed by braces.  Braces are
missing from the indicated aggregate initializer.  Supply the missing
braces for the initializer list.
```


Controlling the use of messages

C.2

The following subsections summarize the command-line options that affect the issuing of messages from the compiler.

Command-line message options

C.2.1

The following arguments to the `-h` option are described in more detail beginning on page 22.

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`.

Option	Description
`-h msglevel_`*n*	Specifies the lowest severity level of messages to be issued.
`-h report=`*args*	Generates optimization report messages.

Environment options for messages C.2.2

The following environment variables are used by the UNICOS message system. See the *UNICOS Message System Programmer's Guide*, publication SG–2121, for more information.

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 UNICOS message system.
`MSG_FORMAT`	Controls the format in which you receive error messages.

ORIG_CMD_NAME environment variable
C.2.3

Caution: The ORIG_CMD_NAME environment variable is not part of the UNICOS 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 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-127 newcc: WARNING File = x.c, Line = 1
  No declarations appear within the compilation unit.
```

Because the environment variable ORIG_CMD_NAME is set to newcc, this appears as the command name instead of cc in this message.

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 CF77 compiling system prints a message. The Fortran message will look as though it came from newcc.

Message severity

C.3

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.

Category	Meaning
`SCALAR`	Information about scalar optimizations performed on the compiled code.
`VECTOR`	Information about vectorization optimizations performed on the compiled code.

Common system messages

C.4

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.

Section C

- `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. This error can occur in integer arithmetic because some integer operations are performed with floating-point arithmetic.

Index

E

F

G

H

I

K

L

M

S

T

U

V

W

Reader's Comment Form

Cray C/C++ Reference Manual SR–2179 2.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with Cray Research computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ____________________

Title ____________________

Company ____________________

Telephone ____________________

Today's date ____________________

Address ____________________

City ____________________

State/Country ____________________

Zip code ____________________

Electronic mail address ____________________

NO POSTAGE
NECESSAR Y
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO 6184 ST. PAUL, MN

POST AGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Cut along this line

Reader's Comment Form

Cray C/C++ Reference Manual SR–2179 2.0

Your reactions to this manual will help us provide you with better documentation. Please take a moment to complete the following items, and use the blank space for additional comments.

List the operating systems and programming languages you have used and the years of experience with each.

Your experience with Cray Research computer systems: _____0-1 year _____1-5 year _____5+years

How did you use this manual: _____in a class _____as a tutorial or introduction _____as a procedural guide _____as a reference _____for troubleshooting _____other

Please rate this manual on the following criteria:

	Excellent			Poor
Accuracy	4	3	2	1
Appropriateness (correct technical level)	4	3	2	1
Accessibility (ease of finding information)	4	3	2	1
Physical qualities (binding, printing, illustrations)	4	3	2	1
Terminology (correct, consistent, and clear)	4	3	2	1
Number of examples	4	3	2	1
Quality of examples	4	3	2	1
Index	4	3	2	1

Please use the space below for your comments about this manual. Please include general comments about the usefulness of this manual. If you have discovered inaccuracies or omissions, please specify the number of the page on which the problem occurred.

Name ____________	Address ____________
Title ____________	City ____________
Company ____________	State/Country ____________
Telephone ____________	Zip code ____________
Today's date ____________	Electronic mail address ____________

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO 6184 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY RESEARCH, INC.

ATTN: Software Publications Group
655 LONE OAK DR BLDG F
EAGAN MN 55121-9957

Fold

Cut along this line