January 1995
Order Number: 313151-001

Paragon System

C++ Compiler User's Guide

Intel® Corporation

WARNING

Some of the circuitry inside this system operates at hazardous energy and electric shock voltage levels. To avoid the risk of personal injury due to contact with an energy hazard, or risk of electric shock, do not enter any portion of this system unless it is intended to be accessible without the use of a tool. The areas that are considered accessible are the outer enclosure and the area just inside the front door when all of the front panels are installed, and the front of the diagnostic station. There are no user serviceable areas inside the system. Refer any need for such access only to technical personnel that have been qualified by Intel Corporation.

CAUTION

This equipment has been tested and found to comply with the limits for a Class A digital device, pursuant to Part 15 of the FCC Rules. These limits are designed to provide reasonable protection against harmful interference when the equipment is operated in a commercial environment. This equipment generates, uses, and can radiate radio frequency energy and, if not installed and used in accordance with the instruction manual, may cause harmful interference to radio communications. Operation of this equipment in a residential area is likely to cause harmful interference in which case the user will be required to correct the interference at his own expense.

LIMITED RIGHTS

Preface

This manual describes the Paragon™ system C++ compiler. This manual assumes that you are an application programmer proficient in the C++ language and the UNIX operating system. This manual describes how to use the C++ compiler and provides general information about the implementation of the C++ language supported. The manual does not attempt to teach the C++ programming language.

Organization

Chapter 1	Introduces the Paragon system software development environment and shows how to create executable files from C++ source code. This chapter contains enough information to get you started creating executable files for the Paragon system.
Chapter 2	Describes **iCC**, the command for compiling, assembling, and linking C++ source code for execution on the Paragon system.
Chapter 3	Gives you a strategy for using the compiler's optimization features to help maximize the single-node performance of your programs.
Chapter 4	Tells how to use the compiler's function inliner.
Chapter 5	Describes the inter-language calling conventions for C++, C, and Fortran.
Chapter 6	Provides general information about the implementation of the C++ language that the C++ compiler accepts.
Chapter 7	Describes the C++ libraries.

Chapter 8	Describes how C++ templates are generated.
Chapter 9	Describes how C++ name mangling is implemented.
Appendix A	Lists the error messages generated by the compiler, indicating each message's severity and, where appropriate, the probable cause of the error and how to correct it.
Appendix B	Contains reference manual pages for the Paragon software development commands.

Compatibility and Conformance to Standards

Since there is no current ANSI standard for C++, the Paragon system C++ compiler produces code that for the most part conforms to *The Annotated C++ Reference Manual*, with modifications supporting the current draft of the ANSI C++ X3J16/WG21 working paper. The section "C++ Dialect Supported" on page 5-1 provides more details on what is and is not supported in the language.

Notational Conventions

This manual uses the following notational conventions:

Bold Identifies command names and switches, system call names, reserved words, and other items that must be entered exactly as shown.

Italic Identifies variables, filenames, directories, processes, user names, and writer annotations in examples. Italic type style is also occasionally used to emphasize a word or phrase.

`Plain-Monospace`
Identifies computer output (prompts and messages), examples, and values of variables. Some examples contain annotations that describe specific parts of the example. These annotations (which are not part of the example code or session) appear in *italic* type style and flush with the right margin.

`Bold-Italic-Monospace`
Identifies user input (what you enter in response to some prompt).

Bold-Monospace
Identifies the names of keyboard keys (which are also enclosed in angle brackets). A dash indicates that the key preceding the dash is to be held down *while* the key following the dash is pressed. For example:

<Break> **<s>** **<Ctrl-Alt-Del>**

[ ] Surround optional items.

... Indicate that the preceding item may be repeated.

| Separates two or more items of which you may select only one.

{ } Surround two or more items of which you must select one.

Applicable Documents

For more information about the C++ language, refer to *The Annotated C++ Reference Manual* by Margaret Ellis and Bjarne Stroustrup. This book is provided with the Paragon system C++ Compiler as an accompanying reference manual.

For more information about Paragon system manuals, refer to the *Paragon™ System Technical Documentation Guide*.

Comments and Assistance

Intel Scalable Systems Division is eager to hear of your experiences with our new software product. Please call us if you need assistance, have questions, or otherwise want to comment on your Paragon system.

U.S.A./Canada Intel Corporation
Phone: 800-421-2823
Internet: support@ssd.intel.com

Intel Corporation Italia s.p.a.
Milanofiori Palazzo
20090 Assago
Milano
Italy
1678 77203 (toll free)

France Intel Corporation
1 Rue Edison-BP303
78054 St. Quentin-en-Yvelines Cedex
France
0590 8602 (toll free)

Intel Japan K.K.
Scalable Systems Division
5-6 Tokodai, Tsukuba City
Ibaraki-Ken 300-26
Japan
0298-47-8904

United Kingdom Intel Corporation (UK) Ltd.
Scalable Systems Division
Pipers Way
Swindon SN3 IRJ
England
0800 212665 (toll free)
(44) 793 491056 (*answered in French*)
(44) 793 431062 (*answered in Italian*)
(44) 793 480874 (*answered in German*)
(44) 793 495108 (*answered in English*)

Germany Intel Semiconductor GmbH
Dornacher Strasse 1
85622 Feldkirchen bei Muenchen
Germany
0130 813741 (toll free)

World Headquarters
Intel Corporation
Scalable Systems Division
15201 N.W. Greenbrier Parkway
Beaverton, Oregon 97006
U.S.A.
(503) 629-7600 (Monday through Friday, 8 AM to 5 PM Pacific Time)
Fax: (503) 629-9147

Table of Contents

Chapter 1
Getting Started

The Software Development Environment 1-1

Compiler Driver 1-3

Execution Environments 1-4

Running on a Single Node 1-4

Running on Multiple Nodes 1-4

Example Driver Command Lines 1-5

Filename Conventions 1-6

Input Files 1-6

Output Files 1-7

Chapter 2
The iCC Driver

Invoking the Driver 2-1

Controlling the Driver 2-3

Specific Passes and Options 2-4

Preprocess Only 2-5

Preprocess and Compile Only 2-5

Preprocess, Compile, and Assemble Only 2-5

Add and Remove Preprocessor Macros 2-6

Controlling the Compilation Step 2-6

Specific Actions 2-7

Location of Include Files 2-11

Optimization Level 2-12

Generating Debug Information 2-12

Controlling the Link Step 2-13

Stripping Symbols 2-13

Generating a Relinkable Object File 2-13

Producing a Link Map 2-14

Linker Libraries 2-14

Controlling Mathematical Semantics 2-14

Controlling the Driver Output 2-15

Executable for Multiple Nodes 2-16

Name of Executable File 2-16

Verbose Mode 2-16

Built-in Math Functions 2-17

Chapter 3
Using the Inliner

Compiler Inline Switch 3-1

Restrictions on Inlining 3-2

Diagnostics During Inlining 3-2

Examples 3-3

Chapter 4
Inter-Language Calling

Inter-Language Calling Considerations 4-1

Functions and Subroutines 4-2

Upper and Lower Case Conventions 4-2

Underscore 4-3

Compatible Data Types 4-3

Fortran Named Common Blocks 4-4

Argument Passing and Return Values 4-4

Passing By Value (%VAL) 4-5

Character Return Values 4-5

Complex Return Values 4-6

Array Indexes 4-6

Examples of Inter-Language Calling 4-7

C++ Calling C 4-7

C Calling C++ 4-7

Fortran Calling C++ 4-8

C++ Calling Fortran 4-9

Chapter 5
C++ Language Considerations

C++ Dialect Supported 5-1
ARM vs. X3J16/WG21 5-1
Anachronisms Accepted 5-3
Extensions Accepted 5-4
Cfront 2.1 Compatibility Mode 5-5
Cfront 2.1/3.0 Compatibility Mode 5-8

Extensions 5-9

Implementation-Defined Behavior 5-10

Data Types 5-11
Scalars 5-11
Alignment of Scalars 5-12
Aggregate Data Types 5-12
Class and Object Data Layout 5-13
Structure Alignment 5-14
Bit-field Alignment 5-16
Other Type Keywords 5-17

Chapter 6
Libraries

Compiler Libraries 6-1
Linking to the Math Library 6-2
Including the Math Header File 6-3
Math Functions 6-3
Standard C Library Calls 6-4
Non-Standard C Library Calls 6-5
Setting the SBRK Size 6-5

The Iostream Library 6-5
Iostreams Header Files 6-5
Iostreams Usage and Class Hierarchy 6-6
Using iostreams 6-6
Class ios 6-7
Class ios Enumerated Types 6-8
io_state enumc 6-8
open_mode enum 6-8
seek_dir enum 6-8
Class ios Constructor and Destructor 6-9
ios 6-9
~ios 6-9
Class ios – Public Functions 6-9
bad 6-9
bitalloc 6-10
clear 6-10
eof 6-10
fail 6-10
fill 6-10
flags 6-11
good 6-11
iword 6-11
precision 6-11
pword 6-11
rdbuf 6-12
rdstate 6-12
setf 6-12
skip 6-12
sync_with_stdio 6-12
tie 6-13
unsetf 6-13
width 6-13
xalloc 6-13

Operators 6-14
Class istream 6-14
Class istream constructor and destructor 6-14
istream 6-14
~istream 6-14
Class istream – Public Functions 6-14
gcount 6-14
get 6-15
getline 6-15
ignore 6-16
ipfx 6-16
peek 6-16
putback 6-16
read 6-16
seekg 6-17
sync 6-17
tellg 6-17
Class ostream 6-17
Class istream constructor and destructor 6-17
ostream 6-17
~ostream 6-18
Class ostream – Public Functions 6-18
flush 6-18
opfx 6-18
osfx 6-18
put 6-18
seekp 6-18
tellp 6-19
write 6-19
ostream Operators 6-19

Class iostream 6-19
iostream Constructor and Destructor 6-20
iostream 6-20
~iostream 6-20
fstream Constructor and Destructor 6-20
fstream 6-20
~fstream 6-21
ofstream Constructor and Destructor 6-21
ofstream 6-21
~oftream 6-21
ifstream Constructor and Destructor 6-22
ifstream 6-22
~ifstream 6-22
Class fstream, ifstream, ofstream – Public functions 6-22
attach 6-22
close 6-23
open 6-23
rdbuf 6-23
setbuf 6-23
Class ostream_withassign 6-23
ostream_withassign 6-23
~ostream_withassign 6-24
Class iostream_withassign 6-24
iostream_withassign 6-24
~iostream_withassign 6-24
Class istream_withassign 6-24
istream_withassign 6-24
~istream_withassign 6-24

Class istrstream 6-25
Class istrstream Constructors and Destructor 6-25
istrstream 6-25
~istrstream 6-25
Class istrstream – Public functions 6-25
rdbuf 6-25
Class ostrstream 6-25
Class ostrstream constructor and destructor 6-26
ostrstream 6-26
~ostrstream 6-26
Class ostrstream – Public Functions 6-26
pcount 6-26
rdbuf 6-26
str 6-27
Class strstream 6-27
Class strstream constructor and destructor 6-27
strstream 6-27
~strstream 6-27
Class strstream – Public Functions 6-27
str 6-27
Class stdiostream 6-28
Class stdiostream constructor and destructor 6-28
stdiostream 6-28
~stdiostream 6-28
Class streambuf 6-28
Class streambuf constructor and destructor 6-28
streambuf 6-28
~streambuf 6-29

Class streambuf – Public Functions 6-29
in_avail 6-29
out_waiting 6-29
sbumpc 6-29
setbuf 6-29
seekoff 6-30
seekpos 6-30
sgetc 6-30
sgetn 6-30
snextc 6-30
sputbackc 6-31
sputc 6-31
sputn 6-31
stossc 6-31
sync 6-31
Class filebuf 6-32
Class filebuf Constructors and Destructors 6-32
filebuf 6-32
~filebuf 6-32
Class filebuf Member Functions 6-32
attach 6-32
close 6-33
fd 6-33
is_open 6-33
open 6-33
Class strstreambuf 6-33
Class strstreambuf—Constructors and Destructor 6-33
strstreambuf 6-33
strstreambuf 6-34
Class strstreambuf - Public Functions 6-35
freeze 6-35
str 6-35
Class stdiobuf 6-35

Manipulators 6-35
dec 6-35
hex 6-36
oct 6-36
ws 6-36
setw 6-36
setfil 6-36
setprecision 6-36
setiosflags 6-37
resetiosflags 6-37
endl 6-37
ends 6-37
flush 6-37

The C++ Complex Math Library 6-38
Complex Constructor and Destructor 6-38
complex 6-38
~complex 6-39
Complex Arithmetic Operators 6-39
Complex Public Functions 6-40
abs 6-40
arg 6-40
conj 6-40
cos 6-40
cosh 6-40
exp 6-40
imag 6-41
log 6-41
norm 6-41
polar 6-41
pow 6-41

real 6-42
sin 6-42
sinh 6-42
sqrt 6-42
Input and Output Using Complex Values 6-42
Error Handling 6-43

Chapter 7
Template Instantiation

Command Line Control 7-2

Automatic Template Instantiation 7-3

Implicit Inclusion 7-5

Chapter 8
C++ Name Mangling

Types of Mangling 8-2

Mangling Summary 8-3
Type Name Mangling 8-3
Nested Class Name Mangling 8-3
Local Class Name Mangling 8-3
Template Class Name Mangling 8-4

Appendix A
Compiler Error Messages

Appendix B
Manual Pages

AR860 ..... B-3

AS860 ..... B-5

DUMP860 ..... B-7

ICC B-9

LD860 ..... B-24

MAC860 ..... B-29

NM860 ..... B-30

SIZE860 ..... B-32

STRIP860 ..... B-34

List of Illustrations

Figure 5-1.Natural Alignment 5-15
Figure 5-2.Quad Alignment 5-16

List of Illustrations

List of Tables

Table 1-1. Software Development Commands 1-2
Table 1-2. Stop After Options, Inputs, and Outputs 1-7

Table 2-1. Summary of iCC Driver Switches 2-1

Table 4-1. Fortran and C++ Date Type Compatibility 4-3
Table 4-2. Fortran and C++ Representation of COMPLEX Data Type 4-4

Table 5-1. Sizes and Alignments of Data Types 5-10
Table 5-2. Scalar Data Types 5-11
Table 5-3. Floating-Point Data Type Ranges 5-12

Table 6-1. Math Functions by Operation and Arguments 6-3
Table B-1. Commands Discussed in This Appendix B-2

List of Tables

Getting Started 1

This chapter introduces the Paragon™ system software development environment and shows how to create executable files from C++ source code.

This chapter contains enough information to get you started using the compiler driver to create executable files from C++ source code.

The Software Development Environment

The Paragon system includes a complete set of commands for compiling, linking, executing, and debugging parallel applications. These commands are available in two different software development environments:

- The *cross-development environment* runs both on the Paragon system and on supported workstations.
- The *native development environment* runs only on the Paragon system itself.

Table 1-1. lists the commands in the two software development environments.

Table 1-1. Software Development Commands

Name in Cross-Development Environment	Name in Native Environment	Description
ar860	**ar**	Manages object code libraries
as860	**as**	Assembles i860™ source code
dump860	**dump860**	Dumps object files
iCC	**CC**	Compiles C++ programs
ld860	**ld**	Links object files
mac860	**mac**	Preprocesses assembly-language programs
nm860	**nm**	Displays symbol table (name list) information
size860	**size**	Displays section sizes of object files
strip860	**strip**	Strips symbol information from object files

With minor exceptions, these commands work the same in both environments and on all supported hardware platforms. The biggest difference between the two environments is the names of the commands, as shown in Table 1-1.. Where other differences exist, they are noted in Appendix B.

NOTE

This manual uses the cross-development names for these commands. However, except where noted, all discussions of the cross-development command names apply equally to the corresponding native command names.

This manual gives complete information on the compiler and provides manual pages for the other commands shown in Table 1-1..

Compiler Driver

The Paragon system C++ driver provides an interface to the compiler, assembler, and linker that makes it easy to produce executable files from C++ source code. For example:

- It automatically sets appropriate compiler, assembler, and linker switches.
- It lets you pass switches directly to the assembler and linker. All functionality of the **as860** assembler and **ld860** linker is available through the driver.
- It lets you stop after the preprocessor, compiler, assembler, or linker steps.
- It lets you retain intermediate files.

The driver creates an executable file for execution on a Paragon system node.

The **iCC** command invokes the C++ driver. For example, the following command line compiles, assembles, and links the C++ source code in the file *myprog.C* (using the default driver switches) and leaves an executable version of the program in the file *a.out*:

```
% iCC myprog.C
```

To translate and link a C++ program, **iCC** does the following:

- Runs the C++ front end and creates a temporary intermediate file (run **icpp1**).
- Runs the back end which reads the binary intermediate file produced by the front end and generates an assembly language file (run **icpp2**).
- Assemble the assembly file to create a coff object file (run **as860**).
- Link in the appropriate startup files and library routines, as well as the application program (run **ld860**).
- Run **nm860** on the linker output file and pipe the output through **imunch**. The command **imunch** is a utility that generates a C program that will call all required start-up initialization routines.

  imunch looks for module level constructor and destructor functions that must be called at program startup time. **imunch** outputs a temporary C language file which contains arrays of function pointers pointing to the module level constructor/destructor functions.
- Compile and assemble the temporary file produced by the prior step.
- Do a final link which includes the temporary module consisting of the tables of module level constructor/destructor calls that need to be made at program startup time.

Additional steps are taken if templates are instantiated. Chapter 2 describes the **iCC** driver in detail.

Execution Environments

The Paragon System software tools can create executable files for execution on one Paragon system node or multiple nodes.

Running on a Single Node

By default, the **iCC** driver creates a file for execution on a single node. For example, the following command line compiles *myprog.C* to the executable *a.out*:

```
% iCC myprog.c
```

When you run the resulting executable by typing **a.out** on the Paragon system, it runs on one node in the service partition.

Running on Multiple Nodes

To run a program on multiple nodes, you must use calls from the library *libnx.a*. This library contains the calls that you use to start processes on multiple nodes and communicate with processes running on other nodes. (All of the calls in *libnx.a* are described in the *Paragon™ System C Calls Reference Manual.*)

The **iCC** driver does not automatically search *libnx.a*. To search *libnx.a*, you can use either the **-nx** or **-lnx** switch when linking:

- The **-nx** switch links in *libnx.a*, *libmach.a*, and *options/autoinit.o* and creates an executable that automatically starts itself on multiple nodes when invoked. For example, the following command line compiles *myprog.C* to the executable *a.out*:

  ```
  % iCC -nx myprog.C
  ```

  When you run the resulting executable by typing **a.out** on the Paragon system, it runs on all the nodes in your default partition. You can use the command line switches and environment variables described in the *Paragon™ System User's Guide* to control its execution characteristics.

- The **-lnx** switch links in *libnx.a* but you should use the **-nx** switch if your program is going to run on multiple nodes. For example, the following command line compiles *myprog.C* to the executable *a.out*:

  ```
  % iCC myprog.C -lnx
  ```

  Note that **-lnx** must appear *after* the filenames of any source or object files that use calls from *libnx.a*.

Example Driver Command Lines

The following example command lines show how to use the **iCC** driver to perform typical tasks. See Chapter 2 for complete information on using the driver and its switches.

- Compile and link for a single Paragon system node, leaving the executable in a file called *x*:

```
% iCC -o x x.C
```

- Compile and link for multiple nodes with automatic start-up:

```
% iCC -nx -o x x.C
```

- Same as above, but include the C math library (**-lm**):

```
% iCC -nx -o x x.C -lm
```

- Compile source file *x.C* and link it together with object file *y.o* and library *mylib.a*:

```
% iCC -o x x.C y.o mylib.a
```

- Compile and link in *libnx.a*:

```
% iCC -o x x.C -lnx
```

- Compile, but skip assemble and link steps (**-S**); leaves assembly language output in file *x.s*:

```
% iCC -S x.C
```

- Compile and assemble, but skip link step (**-c**); leaves object output in file *x.o*:

```
% iCC -c x.C
```

- Compile and assemble with optimizations:

```
% iCC -c -O2 x.C          (level 2 - global optimizations only)
% iCC -c -O3 x.C          (level 3 - adds software pipelining)
% iCC -c -O3 -Mvect x.C   (level 3 optimizations plus vectorization)
```

Filename Conventions

The C++ compiler uses the filenames that you specify on the command line to find and to create input and output files. This section describes the input and output filename conventions for the phases of the compilation process.

Input Files

You can specify C++ source files, assembly-language files, preprocessed C source files, C source files, object files, and libraries as inputs on the **iCC** command line. The driver determines the type of each input file by examining the filename extension. The driver uses the following conventions:

filename.a	A library of object files.
filename.C	A C++ source file that can contain macros and preprocessor directives.
filename.cpp	A C++ source file that can contain macros and preprocessor directives.
filename.cc	A C++ source file that can contain macros and preprocessor directives.
filename.c	A C++ source file that can contain macros and preprocessor directives.
filename.o	An object file.
filename.s	An assembly-language file.

The driver passes files with *.o* and *.a* extensions to the linker and *.s* files to the assembler. Input files with unrecognized extensions or no extensions are also passed to the linker.

Any input files not needed for a particular phase of processing are not processed. For example, if on the command line you use an assembly-language file (*filename*.s) and the **–S** option to stop before the assembly phase, the compiler takes no action on the assembly-language file. Processing stops after compilation and the assembler does not run (in this case compilation must have been completed in a previous pass that created the *.s* file). Refer to Chapter 2 for information on the **–S** option.

In addition to specifying primary input files, files with *.cc*, *.cpp*, *.c*, or *.C* extensions on the command line, you can insert text from include files using the **#include** preprocessor directive. An example of an include file is a library header file which contains declarations used in many different modules (for example *iostream.h*).

When linking a program module with a library, the linker extracts only those library modules that the program needs. For more information about libraries, refer to Chapter 6.

Output Files

By default, **iCC** places executable output in the file *a.out*. You can use the **–o** option to specify a different output file name.

If you use the **–P**, **–S** or **–c** option, the compiler produces a file containing the output of the last phase that completes for each input file, as specified by the option. Using these options, the output file will be a preprocessed source file, an assembly-language file, or an unlinked object file respectively. Similarly, the **–E** option does not produce a file, but displays the preprocessed source file on the standard output. Also, with these options, the **–o** option is valid only if you specify a single input file. If no errors occur during processing, you can use the files created by these options as input to a future invocation of **iCC**. Table 1-2 lists the *stop after* options and the output files that **iCC** creates when you use these options. Note that there are additional steps not covered here to create usable executable files.

Table 1-2. Stop After Options, Inputs, and Outputs

Option	Stop After	Input Files to iCC	Output From iCC
-P	preprocessing	source files	preprocessed files (.i)
-E	preprocessing	source files	preprocessed files to standard out
-S	compilation	C source files preprocessed files	assembly-language files (.s)
-c	assembly	C source files preprocessed files assembly-language files	unlinked object files (.o)
none	linking	C source files preprocessed files assembly-language files object files libraries	executable files (*a.out*)

If you specify multiple input files or do not specify an object filename, the compiler uses the input filenames to derive corresponding default output filenames of the following form, where *filename* is the input filename without its extension:

filename.i indicates a preprocessed file (using the **–P** option).

filename.o indicates an object file from the **–c** option.

filename.s indicates an assembly-language file from the **–S** option.

Note

Unless you specify otherwise, the destination directory for any output file is the current working directory. If the file exists in the destination directory, the compiler overwrites it.

The iCC Driver 2

This chapter describes **iCC**, the driver for compiling, assembling, and linking C++ source code for execution on the Paragon™ system. On the Paragon system, this driver is also available by the name **CC**.

The following sections tell how to invoke **iCC** and how to control its inputs, processing, and outputs.

Invoking the Driver

The **iCC** driver is invoked by the following command line:

```
icc [switches] source_file...
```

where:

switches Is zero or more of the switches listed in Table 2-1.. Note that case is significant in switch names.

source_file Is the name of the file that you want to process. **iCC** bases its processing on the suffixes of the files it is passed. Refer to the section "Filename Conventions" in Chapter 1 for a description of the file name conventions.

Table 2-1. Summary of iCC Driver Switches (1 of 3)

-A	Accept the proposed version of ANSI C++.
-b	Compile with cfront compatibility, version 2.1.
-b3	Compile with cfront compatibility, version 3.0.
-c	Skip link step; compile and assemble only (to *file*.o for each *file*.c).
-C	Preserve comments in preprocessed C source files (implies **-E**).
-dryrun	Show but do not execute commands created by the driver.
-D*name*[=*def*]	Define preprocessor symbol *name* to be *def*.

Table 2-1. Summary of iCC Driver Switches (2 of 3)

-e	Set error limit.
-E	Preprocess every file to *stdout*.
-flags	Display a list of all valid driver options.
-g	Compile for debugging. Synonymous with **-Mdebug -O0 -Mframe -dwarf**.
-help	Display a list of all valid driver options.
-I*directory*	Add *directory* to include file search path.
-K*option*	Request special mathematical semantics (**ieee**, **ieee=enable**, **ieee=strict**, **noieee**, **trap=fp**, **trap=align**).
-l*library*	Load **lib***library***.a** from library search path (passed to the linker).
-L*directory*	Add *directory* to library search path (passed to the linker).
-m	Generate a link map (passed to the linker).
-M*option*	Request special compiler actions (**anno**, **[no]bounds**, **[no]dalign**, **[no]debug**, **[no]depchk**, **[no]frame**, **[no]func32**, **info**, **inline**, **keepasm**, **[no]longbranch**, **nostartup**, **nostddef**, **nostdinc**, **nostdlib**, **[no]perfmon**, **[no]quad**, **[no]reentrant**, **safeptr**, **[no]signextend**, **[no]streamall**, **[no]stride0**, **vect**, **[no]xp**).
-nostdinc	Remove the default include directory from the include files search path.
-nx	Create executable Paragon System application for multiple nodes.
-o*file*	Use *file* as name of output file.
-O[*level*]	Set optimization *level* (**0**, **1**, **2**, **3**, **4**).
-P	Preprocess only (to *file*.i for each *file*.c).
-r	Generate a relinkable object file (passed to the linker).
-rc	Specify the name of the driver configuration file.
-.suffix	Use with **-P** to save intermediate file in a file with the specified suffix.
-s	Strip symbol table information (passed to the linker).
-S	Skip assemble and link step; compile only (to *file*.s for each *file*.c).
-show	Display the driver configuration parameters after startup.
-t *arg*	Control instantiation of template functions.
-time	Print execution times for the compilation steps.
-U*name*	Remove initial definition of *name* in preprocessor.

Table 2-1. Summary of iCC Driver Switches (3 of 3)

-u *symbol*	Initialize the symbol table with *symbol*, which is undefined for the linker. An undefined *symbol* triggers loading of the first member of an archive library.
-v	Print the entire command line for assembler, linker, etc. as each is invoked in verbose mode.
-V	Print the version banner for assembler, linker, etc. as each is invoked.
-VV	Displays the driver version number and the location of the online C++ compiler release notes, but performs no compilation.
-W*pass*,*option*[,*option*...]	Pass *option*s to *pass* (**c**, **0**, **a**, **p**, **l**, **n**, **m**).
-w	Do not print warning messages.
-X	Generate cross-reference information and place output in the specified file.
-Y*pass*,*directory*	Look in *directory* for *pass* (**0**, **a**, **l**, **S**, **I**, **L**, **U**, **P**).

The rest of this chapter discusses these switches in more detail.

Controlling the Driver

The following switches let you control how the driver processes its inputs:

-W Pass specified options to specified tool.

-Y Look in specified directory for specified tool.

-E Skip compile, assemble, and link step; preprocess only (output to *stdout*).

-P Skip compile, assemble, and link step; preprocess only (output to *file*.i).

-S Skip assemble and link step; compile only (output to *file*.s).

-c Skip link step; compile and assemble only (output to *file*.o).

-D Define (create) preprocessor macro.

-U Undefine (remove) preprocessor macro.

Specific Passes and Options

The following switch lets you pass options to specific passes (tools):

-W*pass*,*option*[,*option*...]

where:

pass Is one of the following:

c	C++ front-end.
0 (zero)	C++ back-end.
a	Assembler.
p	Prelinker.
l	Linker.
n	Symbol table lister.
m	Muncher.

option Is a comma-delimited string that is passed as a separate argument.

The following switch lets you tell the driver where to look for a specific pass:

-Y*pass*,*directory*

where *pass* is one of the following:

c	Search for the C++ front-end in *directory*.
0 (zero)	Search for the C++ back-end in *directory*.
a	Search for the assembler executable in *directory*.
l	Search for the linker executable in *directory*.
S	Search for the start-up object files in *directory*.
I	Set the compiler's standard include directory to *directory*.

L Set the first directory in the linker's library search path to *directory* (passes **-YL***directory* to the linker).

U Set the second directory in the linker's library search path to *directory* (passes **-YU***directory* to the linker).

See the **ld860** manual page in Appendix B for more information on the **-YL**, **-YU**, and **-YP** switches.

Preprocess Only

By default, the driver preprocesses, compiles, assembles, and links each input file. However, the following switches suppress the compile, assemble, and link steps:

-E After preprocessing every input file, regardless of suffix, send the result to *stdout*. No compilation, assembly, or linking is performed.

-C After preprocessing each *file*.c, send the result to *stdout* (like **-E**), but do not remove comments during preprocessing.

-P After preprocessing each *file*.c, send the result to a file named *file*.i.

Preprocess and Compile Only

By default, the driver preprocesses, compiles, assembles, and links each input file. However, the following switch tells the driver to suppress the assemble and link steps and produce an assembler source file:

```
-S
```

After compiling each *file*.C, the resulting assembler source file is sent to a file named *file*.s.

Preprocess, Compile, and Assemble Only

By default, the driver preprocesses, compiles, assembles, and links each input file. However, the following switch tells the driver to suppress the link step:

```
-c
```

After assembling each *file*.C, the output is sent to a file named *file*.o. If you are compiling a single source file, you can specify a different output file name with the **-o** switch.

Add and Remove Preprocessor Macros

The following command line switches let you predefine preprocessor macros and undefine predefined preprocessor macros:

NOTE

C++ or ANSI C predefined macros can be defined and undefined on the command line, but not with **#define** and **#undefine** directives in the source.

-D*name*[=*def*] Define *name* to be *def* in the preprocessor. If *def* is missing, it is assumed to be empty. If the "=" sign is missing, then *name* is defined to be the string 1 (one).

-U*name* Remove any initial definition of *name* in the preprocessor. (See also the **nostddef** option of the **-M** switch.)

Because all **-D** switches are processed before all **-U** switches, the **-U** switch overrides the **-D** switch.

The **-U** switch affects only *predefined* preprocessor macros, not macros defined in source files. The following macro names are predefined: **__cplusplus**, **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, **__STDC__**, **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, and **__NODE** (**__NODE** is only defined when compiling with **-nx**).

Note that some of these macro names begin and/or end with *two* underscores.

Controlling the Compilation Step

The following switches let you control the compilation step:

-M*option* Request special compiler actions.

-I Add a directory to include file search path.

-O Set the optimization level.

-g Include symbolic debug information in the output file (synonymous with **-Mdebug -O0 -Mframe -dwarf**).

Specific Actions

The following command line switch lets you request specific actions from the compiler:

```
-Moption
```

where *option* is one of the following (an unrecognized *option* is passed directly to the compiler, which often removes the need for the **-W0** switch):

anno Produce annotated assembly files, where source code is intermixed with assembly language. **-Mkeepasm** or **-S** should be used as well.

[no]bounds [Don't] enable array bounds checking (default **-Mnobounds**). With **-Mbounds** enabled, bounds checking is not applied to subscripted pointers or to externally-declared arrays whose dimensions are zero (**extern arr[]**). Bounds checking is not applied to an argument even if it is declared as an array. If an array bounds checking violation occurs when a program is executed, an error message describing where the error occurred is printed and the program terminates. The text of the error message includes the name of the array, where the error occurred (the source file and line number in the source), and the value, upper bound, and dimension of the out-of-bounds subscript. The name of the array is not included if the subscripting is applied to a pointer.

[no]dalign [Don't] align **double**s in structures on double-precision boundaries (default **-Mdalign**). **-Mnodalign** may lead to data alignment exceptions.

[no]debug [Don't] generate symbolic debug information (default **-Mnodebug**). If **-Mdebug** is specified with an optimization level greater than zero, line numbers will not be generated for all program statements. **-Mdebug** increases the object file size.

[no]depchk [Don't] check for potential data dependencies exist (default **-Mdepchk**). This is especially useful in disambiguating unknown data dependencies between pointers that cannot be resolved at compile time. For example, if two floating-point array pointers are passed to a function and the pointers never overlap and thus never conflict, then this switch may result in better code. The granularity of this switch is rather coarse, and hence the user must use precaution to ensure that other *necessary* data dependencies are not overridden. Do not use this switch if such data dependencies do exist. **-Mnodepchk** may result in incorrect code; the **-Msafeptr** switch provides a less dangerous way to accomplish the same thing.

[no]frame [Don't] include the frame pointer (default **-Mnoframe**). Using **-Mnoframe** can improve execution time and decrease code, but makes it impossible to get a call stack traceback when using a debugger.

[no]func32 [Don't] align functions on 32-byte boundaries (default **-Mfunc32**). **-Mfunc32** may improve cache performance for programs with many small functions.

info=[*option*[**,***option*...]]
Produce useful information on the standard error output. The options are:

time or **stat**	Output compilation statistics.
loop	Output information about loops. This includes information about vectorization and software pipelining.
inline	Output information about functions extracted and inlined.

inline=[*option*[**,***option*...]]
Pass options to the function inliner. The *options* are:

levels:*number*	Perform *number* levels of inlining (default 1).

See Chapter 3 for more information on using the compiler's function inliner.

keepasm Keep the assembly file for each source file, but continue to assemble and link the program.

nolist Don't create a listing file (this is the default).

[no]longbranch [Don't] allow compiler to generate **bte** and **btne** instructions (default **-Mlongbranch**). **-Mnolongbranch** should be used only if an assembly error occurs.

nostartup Don't link the usual start-up routine (*crt0.o*), which contains the entry point for the program.

nostddef Don't predefine any system-specific macros to the preprocessor when compiling a C program. (Does not affect ANSI-standard preprocessor macros.) The system-specific predefined macros are **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, and **__NODE** (**__NODE** is only defined when compiling with **-nx**). See also **-U**.

nostdinc Remove the default include directory (*/usr/include* for **CC**, *$(PARAGON_XDEV)/paragon/include* for **iCC**) from the include files search path.

nostdlib Don't link the standard libraries (*libpm.o*, *guard.o*, *libc.a*, *iclib.a*, and *libmach3.a*) when linking a program.

[no]perfmon [Don't] link the performance monitoring module (*libpm.o*) (default **-Mperfmon**). See the *Paragon™ System Application Tools User's Guide* for information on performance monitoring.

[no]quad [Don't] force top-level objects (such as local arrays) of size greater than or equal to 16 bytes to be quad-aligned (default **-Mquad**). Note that **-Mquad** does not affect items within a top-level object; such items are quad-aligned only if appropriate padding is inserted.

[no]reentrant [Don't] generate reentrant code (default **-Mreentrant**). **-Mreentrant** disables certain optimizations that can improve performance but may result in code that is not reentrant. Even with **-Mreentrant**, the code may still not be reentrant if it is improperly written (for example, if it declares static variables).

safeptr=[*option*[,*option*...]]
Override data dependence between C++ pointers and arrays. This is a potentially very dangerous option since the potential exists for code to be generated that will result in unexpected or incorrect results as is defined by the ANSI C++ working draft. However, when used properly, this option has the potential to greatly enhance the performance of the resulting code, especially floating-point oriented loops. Combinations of the *options* can be used.

dummy or **arg** C++ dummy arguments (pointers and arrays) are treated with the same copyin/copyout semantics as Fortran dummy arguments.

auto C++ local or **auto** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.

static C++ **static** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.

global C++ global or **extern** variables (pointers and arrays) are assumed not to overlap or conflict with each other and are independent.

[no]signextend [Don't] sign extend when a narrowing conversion overflows (default **-Msignextend**). For example, if **-Msignextend** is in effect and an integer containing the value 65535 is converted to a **short**, the value of the **short** will be -1. This option is provided for compatibility with other compilers, even though ANSI C specifies that the result of such conversions are undefined. **-Msignextend** will decrease performance on such conversions.

[no]streamall [Don't] stream all vectors to and from cache in a vector loop (default **-Mstreamall**). When **-Mnostreamall** is in effect, the compiler chooses one vector to come directly from or go directly to main memory, without being streamed into or out of cache.

[no]stride0 [Don't] inhibit certain optimizations and allow for stride 0 array references. **-Mstride0** may degrade performance, and should only be used if zero stride induction variables are possible. (default **-Mnostride0**).

vect[=*option*[,*option*...]]

Perform vectorization (also enables **-Mvintr**). If no *option*s are specified, then all vector optimizations are enabled. The available *option*s are:

cachesize:*number*

This sets the size of the portion of the cache used by the vectorizer to *number* bytes. *Number* must be a multiple of 16, and less than the cache size of the microprocessor (16384 for the i860 XP, 8192 for the i860 XR). In most cases the best results occur when *number* is set to 4096, which is the default (for both microprocessors).

noassoc When scalar reductions are present (for example, dot product), and loop unrolling is turned on, the compiler may change the order of operations so that it can generate better code. This transformation can change the result of the computation due to round-off error. The use of **noassoc** prevents this transformation.

recog Recognize certain loops as simple vector loops and call a special routine.

smallvect[*:number*]
This option allows the vectorizer to assume that the maximum vector length is no greater than *number*. *Number* must be a multiple of 10. If *number* is not specified, the value 100 is used. This option allows the vectorizer to avoid stripmining in cases where it cannot determine the maximum vector length. In doubly-nested, non-perfectly nested loops this option can allow invariant vector motion that would not otherwise have been possible. Incorrect code will result if this option is used, and a vector takes on a length greater than specified.

streamlim:*n* This sets a limit for application of the vectorizer data streaming optimization. If data streaming requires cache vectors of length less than *n*, the optimization is not performed. Other vectorizer optimizations are still performed. The data streaming optimization has a high overhead compared to other loop optimizations, and can be counter-productive when used for short vectors. The *n* specifier is not optional. The default limit is 32 elements if **streamlim** is not used.

transform Perform high-level transformations such as loop splitting and loop interchanging. This is normally not useful without **-Mvect=recog**.

-Mvect with no options means **-Mvect=recog,transform,cachesize:4096**.

[**no**]**xp** [Don't] use i860 XP microprocessor features (default **-Mxp**).

Location of Include Files

The following command line switch lets you add a specified directory to the compiler's search path for include files:

```
-Idirectory
```

where *directory* is the pathname of the directory to be added. If you use more than one **-I** switch, the specified directories are searched in the order they were specified (left to right).

For include files surrounded by angle brackets (<...>), each **-I** directory is searched, followed by the standard include directory. For include files surrounded by double quotes ("..."), the directory containing the file containing the **#include** directive is searched, followed by the **-I** directories, followed by the standard include directory.

Optimization Level

The following command line switch lets you set the optimization level explicitly:

-O[*level*]

where *level* is one of the following:

0	A basic block is generated for each C++ statement. No scheduling is done between statements. No global optimizations are performed.
1	Scheduling within extended basic blocks is performed. Some register allocation is performed. No global optimizations are performed.
2	All level 1 optimizations are performed. In addition, traditional scalar optimizations such as induction recognition and loop invariant motion are performed by the global optimizer.
3	All level 2 optimizations are performed. In addition, software pipelining is performed.
4	All level 3 optimizations are performed, but with more aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file.

If **-O** is used without a *level*, the optimization level is set to 2. If you do not use the **-O** switch, the default optimization level is 1.

NOTE

When compiling an application for debugging, you will get the best results using **-O0**.

Generating Debug Information

To compile for debugging you should use the **-g** compiler switch. The **-g** switch is equivalent to **-Mdebug -Mframe -O0 -dwarf**. These switches have the following effects:

-Mdebug	Generate symbol and line number information.
-Mframe	Generate stack frames on function calls. (Default **-Mnoframe**.) Debugging code without stack frames generated on function calls will result in stack tracebacks that have missing calls when you use the **frame** command.

-O0	Optimization off. If you do not turn optimization off, access to individual source lines will be decreased, and display or modification of variables and registers will probably have unpredictable results.
-dwarf	Generate dwarf format.

You can debug programs not compiled for debugging, but your ability to debug will be very limited. The debugging information generated by **-g** increases the object file size.

Controlling the Link Step

The following switches let you control the link step (they are all passed directly to the linker):

-s	Strip symbol table information.
-r	Generate a relinkable object file.
-m	Produce a link map.
-L	Change the default library search path.
-l	Load a specific library.

Stripping Symbols

The following command line switch strips all symbols from the output object file:

```
-s
```

This results in a smaller object file.

Generating a Relinkable Object File

The following command line switch generates a relinkable object file:

```
-r
```

When you use the **-r** switch, the linker keeps internal symbol information in the resulting object file. This lets you link the object file together with other object files later.

Producing a Link Map

The following command line switch produces a link map on the standard output:

```
-m
```

The link map lists the start address of each section in the object file. To get more information about the object file, use the **dump860** command.

Linker Libraries

The following switch adds a directory to the head of the linker's library search path:

```
-Ldirectory
```

where *directory* is the pathname of a directory that the linker searches for libraries. The linker searches *directory* first (before the default path and before any previously specified **-L** paths).

The following switch tells the linker to use a specific linker library:

```
-llibrary
```

The linker loads the library **lib***library*.**a** from the first library directory in the library search path in which a file of that name is encountered.

See the **ld860** manual page in Appendix B for more information on the linker's library search path.

Controlling Mathematical Semantics

The following command line switch lets you request special mathematical semantics from the compiler and linker:

```
-Koption
```

where *option* is one of the following:

ieee If used while linking, links in a math library that conforms with the IEEE 754 standard.

If used while compiling, tells the compiler to perform **float** and **double** divides in conformance with the IEEE 754 standard.

ieee=enable If used while linking, has the same effects as **-Kieee**, and also enables floating-point traps and underflow traps. If used while compiling, has the same effects as **-Kieee**.

ieee=strict If used while linking, has the same effects as **-Kieee=enable**, and also enables inexact traps. If used while compiling, has the same effects as **-Kieee**.

noieee If used while linking, produces a program that flushes denormals to 0 on creation, which reduces underflow traps. If used together with **-lm**, also links in a version of *libm.a* that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as **INF** and **NaN**, and will trap on such values when encountered.

If used while compiling, tells the compiler to perform **float** and **double** divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the IEEE standard by no more than three units in the last place.

trap=fp If used while compiling, disables kernel handling of floating-point traps. Has no effect if used while linking.

trap=align If used while compiling, disables kernel handling of alignment traps. Has no effect if used while linking.

-Kieee is the default.

Controlling the Driver Output

The following switches let you control the driver's outputs:

-nx Create an executable Paragon system application for multiple nodes.

-o Specify the name of the output file.

-V Print the version banner for each tool (assembler, linker, etc.) as it is invoked.

-VV Display the driver version number and the location of the online release notes. but do not perform any compilation.

-v Print the entire command line for each tool as it is invoked, and invoke each tool in verbose mode (if it has one).

Executable for Multiple Nodes

By default, the **iCC** driver creates an executable for a single node. The following command line switch creates an executable for multiple nodes:

```
-nx
```

The **-nx** switch has three effects:

- If used while compiling, it defines the preprocessor symbol **__NODE**. The program being compiled can use preprocessor statements such as **#ifdef** to control compilation based on whether or not this symbol is defined.
- If used while linking, it links in *libnx.a*, the library that contains all the calls in the *Paragon™ System C Calls Reference Manual*. It also links in *libmach.a* and *options/autoinit.o*.
- If used while linking, it links in a special start-up routine that automatically copies the program onto multiple nodes, as specified by standard command-line switches and environment variables. See the *Paragon™ System User's Guide* for information on these command-line switches and environment variables.

Name of Executable File

By default, the executable file is named *a.out* (or *file*.o if you use the **-c** switch). However, the following command line switch lets you name the file anything you like:

```
-ofile
```

where *file* is the desired name.

Verbose Mode

By default, the driver does its work silently. However, the following command line switch causes the driver to display the version banner of each tool (assembler, linker, etc.) as it is invoked:

```
-V
```

The following command line switch causes the driver to identify itself in more detail than the **-V** switch and display the location of the online compiler release notes. No compilation is performed:

```
-VV
```

The following command line switch causes the driver to display the entire command line that invokes each tool, and to turn on verbose mode (if available) for each tool:

```
-v
```

Built-in Math Functions

The compiler supports the recognition of certain math functions as built-ins. These functions are defined in the file *math.h* with **#define** statements. The **#define** statements are of the form:

#define *routine*(*args*) **__builtin_***routine* (*args*)

routine is the name of a math function, and *args* are the arguments to the function. The following is an example of a **#define** statement that defines the absolute value function as a built-in:

```
#define abs(x)__builtin_abs(x)
```

Having built-in functions provides two benefits:

- Built-in functions allow the vectorizer to recognize vector versions of the functions, if they exist. These vector intrinsics are optimized and provide significant performance improvements for vector operations.
- Built-in functions cause the code for a function to be generated inline, rather than incurring the overhead of a function call.

For functions to be defined as built-ins, the **__PGI** macro must be defined. This macro is defined by default.

The following is a list of the built-in math functions.

abs(*x*)	**fabs**(*x*)	**fabsf**(*x*)
acos(*x*)	**acosf**(*x*)	**asin**(*x*)
asinf(*x*)	**atan**(*x*)	**atanf**(*x*)
atan2(*x*,*y*)	**atan2f**(*x*,*y*)	**cos**(*x*)
cosf(*x*)	**cosh**(*x*)	**coshf**(*x*)
exp(*x*)	**expf**(*x*)	**log**(*x*)
logf(*x*)	**log10**(*x*)	**log10f**(*x*)
pow(*x*,*y*)	**powf**(*x*,*y*)	**sin**(*x*)
sinf(*x*)	**sinh**(*x*)	**sinhf**(*x*)
sqrt(*x*)	**sqrtf**(*x*)	**tan**(*x*)
tanf(*x*)	**tanh**(*x*)	**tanhf**(*x*)

Using the Inliner 3

This chapter describes the compiler's function inlining capability.

Function inlining is a compiler optimization under which the body of a function is expanded in place of a call to the function. This can speed up execution by eliminating the parameter passing and function call and return overhead. Inlining a function body also creates opportunities for other compiler optimizations. Inlining will usually result in larger code size (although in the case of very small functions, code size can actually decrease). Using inlining indiscriminately can result in much larger code size and no increase in execution speed; there may even be a decrease in execution speed.

C++ allows programmers to specify functions as inline by using the keyword **inline**. Functions whose definitions are given in a class declaration are also considered inlinable functions. The C++ compiler takes the inline specification as a hint that the user would like the function expanded inline. The **inline** keyword is a hint to the C++ compiler in the same sense that a **register** declaration is a hint to a C compiler that the programmer would like a variable to reside in a register.

In general, the C++ compiler is not required to inline a function whose definition does not appear in the source file stream before function calls to that function.

When you use the **-Minline** switch during compilation, the compiler first looks in the source files for functions that can be inlined, then replaces calls to those functions with the equivalent code automatically.

Compiler Inline Switch

To invoke the function inliner, use the **-Minline** switch. The compiler performs a special pass on all source files named on the compiler command line. This prepass extracts functions that meet the requirements for inlining and puts them in a temporary inline library for use by the compilation pass.The **-Minline** switch has the following syntax:

-Minline=levels:*n*

where *n* is a level number that represents the number of function calling levels to be inlined. The default number is 1. Using a level greater than 1 indicates that function calls within inlined functions may be replaced with inlined code. This allows the function inliner to automatically perform a sequence of inline and extract processes. Setting *levels* to 0 turns off function inlining.

Using the **–Minline=levels:1** switch causes most user specified inline function calls to be expanded inline (see the following section for restrictions on function inlining). Inlined functions called by other inlined functions may not be expanded inline with the option **levels:1**. Specifying **–Minlines=levels:***n* where *n* is 2 or greater may create more inline expansion. Raising the levels number increases compilation time. Specifying **–Minline=levels:0** suppresses inlining. This speeds up compilation and may make it easier to debug some programs.

Restrictions on Inlining

The following functions cannot be inlined:

- Functions whose return type is a struct data type, or have a struct argument.
- Functions containing switch statements.
- Functions that reference a static variable whose definition is nested within the function.
- Functions that accept a variable number of arguments.

Certain functions can only be inlined into the file that contains their definition:

- Static functions.
- Functions that call a static function.
- Functions that reference a static variable.

Diagnostics During Inlining

For information on inlining, set the diagnostics reporting switch (**-Minfo=inline**).

An additional feature associated with inlining is enhanced compiler error detection. For example:

- If an inlinable function is called with the wrong number of arguments, a warning message is issued and the function is not inlined.
- If an inlinable function is called in a context which assumes that a value is returned, but the body of the function does not contain any statements that set the return value, a severe error is issued.
- If the declaration of an external variable referenced by an inlinable function does not match the declaration in the source file being compiled, a severe error is issued.

Examples

This section contains examples of using the inliner.

The following command line builds an executable file in which inline functions are inlined (the default action of the compiler):

```
$ iCC tests.C
```

The following command line creates an executable file in which inline functions are not inlined

```
$ iCC -Minline=levels:0 tests.C
```

The following command line creates an executable file in which two levels of inlining are performed.

```
$ iCC -Minline=levels:2 tests.C
```

Inter-Language Calling 4

This chapter describes inter-language calling conventions for C, Fortran and C++ programs. The chapter describes how to call a Fortran or C function or subroutine from a C++ program and how to call a C++ function from a Fortran or C program. At the end of the chapter several examples are presented showing inter-language calling.

This chapter includes information on the following topics:

- Functions and subroutines in Fortran, C, and C++.
- Naming and case conversion conventions.
- Compatible data types.
- Argument passing and special return values.
- Arrays and indexes.

Inter-Language Calling Considerations

In general, when argument data types and function return values agree, you can call a C or Fortran function from C++ and likewise, you can call a C++ function from C or Fortran. Where data types for arguments do not agree, for example the Fortran **COMPLEX** type does not have a matching type in C++, it is still possible to provide inter-language calls, but there are no general calling conventions for such cases (you may need to develop special procedures to handle such cases).

Note that if a C++ function contains objects with constructors and destructors, calling such a function from either C or Fortran will not be possible unless the initialization in the main program is performed from a C++ program where constructors and destructors are properly initialized.

In general, you can call a C function from C++ without problems as long as you use the **extern "C"** keyword to declare the C function in the C++ program. This prevents name mangling for the C function name. If you want to call a C++ function from C, likewise you have to use the **extern "C"** keyword to declare the C++ function. This keeps the C++ compiler from mangling the name of the function.

You can use the **__cplusplus** macro to allow a program or header file to work for both C and C++. For example, the following defines in the header file *stdio.h* allow this file to work for both C and C++.

```
#ifndef _STDIO_H
#define _STDIO_H

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
. /* Functions and data types defined... */
.
#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif
```

C++ member functions cannot be declared **extern**, as their names will always be mangled. Therefore, C++ member functions cannot be called from C or Fortran.

Functions and Subroutines

Fortran and C++ define functions and subroutines differently. For a Fortran program calling a C++ program, observe the following return value convention: When the C++ function returns a value, call it from Fortran as a function, otherwise call it as a subroutine.

For a C++ program calling a Fortran function, the call from C++ should return a similar type. Table 4-1 lists compatible types. If the call is to a Fortran subroutine or Fortran function, call it from C++ as a function that returns *void*. The exception to this convention is when a Fortran subroutine has alternate returns; call such a subroutine from C++ as a function returning *int* whose value is the value of the integer expression specified in the alternate **RETURN** statement.

Upper and Lower Case Conventions

By default, all Fortran program names are converted to lower-case. C++ is case sensitive, so upper-case function names stay upper-case. When you use inter-language calling you can either name your C++ functions with lower-case names, or use the **if77** option **–Mupcase** which tells Fortran not to convert to lower-case.

Underscore

When programs are compiled, **if77** appends an underscore to Fortran global names (names of functions, subroutines and common blocks). This mechanism distinguishes Fortran name space from C++ name space. If you call a C++ function from Fortran, you should rename the C++ function by appending an underscore. If you call a Fortran function from C++, you should append an underscore to the Fortran function name in the calling program.

Compatible Data Types

Table 4-1 shows compatible data types between Fortran and C++. Table 4-2 shows how you can represent the Fortran **COMPLEX** type in C++. If you can make your function/subroutine parameters and return values match types, you should be able to use inter-language calling. An exception is that you cannot directly call a **COMPLEX** function from C++. Refer to the section "Complex Return Values" on page 4-6 for details on how to call a **COMPLEX** function indirectly.

Table 4-1. Fortran and C++ Date Type Compatibility

Fortran Type (lowercase)	C++ Type	Size (bytes)
integer*1 x	signed char x	1
character x	char x	1
character*n x	char x[n]	n
double precision	double x	8
real x	float x	4
real*4 x	float x	4
real*8 x	double x	8
integer x	int x	4
integer*4 x	int x	4
integer*2 x	short x	2
logical x	int x	4
logical*4 x	int x	4
logical*2 x	short x	2
logical*1 x	short x	1

Table 4-2. Fortran and C++ Representation of COMPLEX Data Type

Fortran Type (lowercase)	C++ Type	Size (bytes)
complex x	struct {float r,i;} x;	8
complex*8 x	struct {float r,i;} x;	8
double complex x	struct {double dr,di;} x;	16

Fortran Named Common Blocks

A named Fortran common block can be represented in C++ by a structure whose members correspond to the members of the common block. The name of the structure in C++ must have the added underscore. For example the Fortran common block:

```
INTEGER I
COMPLEX C
COMPLEX CD
DOUBLE PRECISION D
COMMON /COM/ i, c, cd, d
```

is represented in C++ with the following equivalent:

```
extern "C" struct {
     int i;
     struct {float real, imag;} c;
     struct {float real, imag;} cd;
     double d;
} com_;
```

Argument Passing and Return Values

In Fortran, arguments are passed by reference. In C++ parameters are passed by value, except for strings and arrays, which are passed by reference. Due to the flexibility provided in C++, you can work around these differences. Generally, solving the parameter passing differences involves intelligent use of the & and * operators in argument passing when C++ calls Fortran and in argument declarations when Fortran calls C++.

For strings declared in Fortran as type **CHARACTER**, an argument representing the length of the string is passed to a calling function. The compiler places the length argument(s) at the end of the parameter list, following the other formal arguments. The length argument is passed by value, not by reference.

Passing By Value (%VAL)

When passing parameters from a Fortran subprogram to a C++ function, it is possible to pass by value using the **%VAL** function. If you enclose a Fortran parameter with **%VAL()**, the parameter is passed by value. For example, the following call passes the integer I and the logical BVAR by value.

```
INTEGER*1    I
LOGICAL*1    BVAR

CALL CPLUSVALUE (%VAL(i), %VAL(BVAR))
```

Character Return Values

The section "Functions and Subroutines" on page 4-2 describes the general rules for return values for C++ and Fortran inter-language calling. There is a special return value to consider. When a Fortran function returns a character, two arguments need to be added at the beginning of the C++ *calling* function's argument list: the address of the return character or characters, and the length of the return character. The following example shows two extra parameters, tmp and 10, supplied by the caller.

```
      CHARACTER*(*) FUNCTION CHF( C1, I)
      CHARACTER*(*) C1
      INTEGER I
      END

extern void chf_();
char tmp[10];
char c1[9];
int i;
chf_(tmp, 10, c1, &i, 9);
```

If the Fortran function is declared to return a character value of constant length, for example **CHARACTER*4 FUNCTION CHF()**, the second extra parameter, representing the length, must still be supplied but is not used.

NOTE

The value of the character function is not automatically NULL-terminated.

Complex Return Values

Fortran complex functions return their values in multiple floating-point registers; consequently, you cannot directly call a Fortran complex function from C++. It is possible for a C++ function to pass a pointer to a memory area to a function, which calls the COMPLEX function and stores the value returned by the complex function. The following example illustrates COMPLEX return values.

```
extern void inter_cf_();
typedef struct {float real, imag;} cplx;
cplx c1;
int 1;
inter_cf_(&c1, &i);
```

```
   SUBROUTINE INTER_CF(C, I)
   COMPLEX C
   COMPLEX CF
   C = CF(I)
   RETURN
   END


   COMPLEX FUNCTION CF(I)
         .
         .
         .
   END
```

Array Indexes

C++ arrays and Fortran arrays use different default initial array index values. By default, C++ arrays start at 0 and Fortran arrays start at 1. If you adjust your array comparisons so that a Fortran second element is compared to a C first element, and adjust similarly for other elements, you should not have problems working with this difference. If this is not satisfactory, you can declare your Fortran arrays to start at zero.

Another difference between Fortran and C++ arrays is the storage method used. Fortran uses column-major order and C++ uses row-major order. For one-dimensional arrays, as you might expect, this poses no problems. For two-dimensional arrays, where there are an equal number of rows and columns, row and column indexes can simply be reversed. For arrays other than single-dimensional arrays, and square two-dimensional arrays, inter-language function mixing is not recommended.

Examples of Inter-Language Calling

The following sections show examples of inter-language calling between C++, C, and Fortran.

C++ Calling C

The following is a simple C function:

```
void func1_cplus(num1, num2, res)
int num1, num2, *res;
{
*res=num1/num2;
}
```

The following example shows the C++ Main calling the C function:

```
extern "C" void func1_cplus();
main()
{
   int a,b,c;
   a=8;
   b=2;
   func1_cplus(a,b,&c);
.
}
```

C Calling C++

The following is a simple C++ function with extern C:

```
extern "C" void func2_cplus(int num1,int num2,int *res);
{
*res=num1/num2;
}
```

Now you can compile the function func1_cplus with **iCC** and link it with your C programs.

```
extern void func2_cplus();
main()
{
   int a,b,c;
   a=8;
   b=2;
   func1_cplus(a,b,&c);
.
.
}
```

Note that you cannot use the **extern "C"** form of declaration for an object's member functions.

Fortran Calling C++

The following is a C++ function that will be called by a Fortran main program. Note that each argument is defined as a pointer, since Fortran passes by reference. Also note that the C++ function name uses all lowercase and a trailing underscore.

```
#define TRUE 0xff
#define FALSE 0
extern "C" {

extern void cplus_func_ (
   char     *bool1,
   char     *letter1,
   int      *numint1,
   int      *numint2,
   float    *numfloat1,
   double   *numdoub1,
   short    *numshort1,
   int      len_letter1) {

   *bool1= TRUE;
   *letter1 = 'v';
   *numint1 = 11;
   *numint2  = -44;
   *numfloat1 = 39.6;
   *numdoub1 = 39.2;
   *numshort1 = 981;

}
}
```

The following Fortran main program calls the C++ function **cplus_func**:

```
      logical*1              bool1
      character              letter1
      integer*4              numint1, numint2
      real                   numfloat1
      double precision       numdoub1
      integer*2              numshor1
      external cfunc
      call cplus_func (bool1, letter1, numint1,
     &    numint2, numfloat1, numdoub1, numshor1)
      write( *, "(L2, A2, I5, I5, F6.1, F6.1, I5)")
     &    bool1, letter1, numint1, numint2, numfloat1,
     &    numdoub1, numshor1
      end
```

To execute the Fortran program **fmain.f**, and call **cfunc_**, create an executable using the following command lines:

```
$ iCC -c cfunc.C
$ if77 cfunc.o fmain.f
```

Executing the *a.out* file should produce the following output:

```
T  v  11  -44 39.6  39.2  981
```

C++ Calling Fortran

The following is a Fortran subroutine that will be called by a C++ function. Note that the subroutine uses all lowercase.

```
      subroutine forts ( bool1, letter1, numint1,
     &    numint2, numfloat1, numdoub1, numshor1)
      logical*1           bool1
      character           letter1
      integer             numint1, numint2
      double precision    numdoub1
      real                numfloat1
      integer*2           numshor1
```

```
bool1 = .true.
letter1 = "v"
numint1 = 11
numint2 = -44
numdoub1 = 902
numfloat1 = 39.6
numshor1 = 299
return
end
```

The following C++ main calls the Fortran subroutine. Note that each call uses the & operator to pass by reference.

```
#include <stdio.h>

extern "C" {
extern  void forts_ ( char *, char *, int *, int *, float *,
double *, short * );
}

main ()
{
   char          bool1, letter1;
   int           numint1, numint2;
   float         numfloat1;
   double        numdoub1;
   short         numshor1;
   forts_(&bool1,&letter1,&numint1,&numint2,&numfloat        1,
&numdoub1,&numshor1, 1);
   printf(" %s %c %d %d %3.1f %.0f
%d\n",bool1?"TRUE":"FALSE",letter1,numint1,
   numint2, numfloat1, numdoub1, numshor1);
}
```

To compile this Fortran subroutine and C++ program, use the following command lines:

```
$ if77 -c forts.f
$ iCC forts.o cmain.C
```

Executing this C main should produce the following output:

```
TRUE  v  11 -44  39.6  902  299
```

C++ Language Considerations 5

This chapter describes the language that the Paragon™ system C++ compiler accepts, extensions to the standard language, and details about the C++ dialect supported and C++ data types.

C++ Dialect Supported

The Paragon system C++ compiler accepts the C++ language as defined by *The Annotated C++ Reference Manual* (ARM) by Ellis and Stroustrup, Addison-Wesley, 1990, including templates and support for the anachronisms described in section 18.3 of the ARM. This is the same language defined by the language reference for ATT's cfront version 3.0.x. The C++ compiler optionally accepts a number of *features* erroneously accepted by cfront version 2.1 or 3.0. Using the **–b** option, **iCC** accepts these features, which may never have been legal C++, but have found their way into some code.

Full support of many C++ variants, accomplished by accepting many language extensions and anachronisms, is provided by command-line options. The C++ compiler provides command-line options that enable the user to specify whether anachronisms and/or cfront 2.1 or 3.0 compatibility features should be accepted. Refer to the following section for details on features that are not part of the ARM but are part of the ANSI C++ working draft X3J16/WG21.

ARM vs. X3J16/WG21

The following features not in the ARM but in the X3J16/WG21 Working Paper are accepted:

- The dependent statement of an **if**, **while**, **do-while**, or **for** is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an **if**, **while**, **do-while**, or **for**, as the first operand of a "?" operator, or as an operand of the "&&", "::", or "!" operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.
- Qualified names are allowed in elaborated type specifiers.

- Use of a global-scope qualifier in member references of the form x.::A::B and p->::A::B.
- 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 not diagnosed as errors.

The following features not in the ARM but in the X3J16/WG21 Working Paper are not accepted:

- Virtual functions in derived classes may not return a type that is the derived-class version of the type returned by the overridden function in the base class.
- **enum** types are not considered to be non-integral types.
- The digraph and macro forms (e.g., certain operators and punctuators are not accepted).
- **wchar_t** is not treated as a keyword.
- It is not possible to define class-specific new and delete routines to be used for arrays of class objects.
- It is not possible to overload operators using functions that take **enum** types and no class types.
- Runtime type identification (RTTI) is not implemented.
- Declarations in tested conditions are not implemented.
- 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.
- Implicit conversion of **T**** to **const T *const *** (and the like) is not yet implemented.
- A cast cannot be used to select one out of a set of overloaded functions when taking the address of a function.
- Type qualifiers on parameter types are not dropped.
- Classes are not assumed to always have constructors, and the distinction between trivial and nontrivial constructors is not implemented.
- **mutable** is not implemented.
- The lifetime of temporaries is not limited to statements (it extends to end of block).
- Namespaces are not implemented.

- **enum** types cannot contain values larger than can be contained in an **int**.
- Type qualifiers are not retained on rvalues (in particular, on function return values).
- Functions that cannot be called because of incorrect reference binding are nevertheless considered in overload resolution.
- **bool** is not implemented.
- New casts (e.g., **reinterpret_cast**) are 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 still the whole surrounding scope, not just the loop.
- Static data member declarations cannot be used to declare member constants.

Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled:

- **overload** is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. This anachronism does not apply to static data members of template classes; they must always be defined.
- Implicit conversion from integral types to enumeration types is allowed.
- The number of elements in an array may be specified in an array delete operation. The value is ignored.
- A single **operator++**() and **operator--**() function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- Assignment to **this** in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the assignment to this configuration parameter is enabled.

- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. This anachronism is not applied to template classes.
- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named f:

```
int f(int); int f(x) char x;  return x;
```

  It will be noted that in C this is code is legal but has a different meaning: a tentative declaration of f is followed by its definition.

Extensions Accepted

The following extensions are accepted in all modes (except when strict ANSI violations are diagnosed as errors; see the **–A** option):

- A friend declaration for a class may omit the class keyword.

```
class A {
  friend B;  // Should be "friend class B"
};
```

- Constants of scalar type may be defined within classes.

```
class A {
  const int size = 10;
  int a[size];
};
```

- In the declaration of a class member, a qualified name may be used.

```
struct A{
    int A::f();  // Should be int f();
}
```

- **operator**()() functions may have default argument expressions. A warning is issued.

- The preprocessing symbol **c_plusplus** is defined in addition to the standard **__cplusplus**.
- In a reference to a class template, the argument corresponding to a non-type parameter is allowed to require a promotion or standard conversion. The ARM requires an exact match.

```
template<float F> class S{};
S<2.0> x;
```

- Trivial conversions involving references, array and function type decay to pointer, and addition of type qualifiers are allowed.

```
template <class T> void f(const T &p) {}
void m() {int i; f(i); }
```

- Parameters that do not involve template types are allowed to require trivial conversions and/or a cast to a base class.

```
struct A {};
struct B : public A {} *p;
template<class T> void f(T, A*);
void m() {f(1, p);}
```

- A reference to a non-const class type can be initialized from a non-lvalue of the proper type or a derived type thereof.

Cfront 2.1 Compatibility Mode

The following extensions are accepted in cfront 2.1 compatibility mode (with the **-b** option), along with the extensions listed in the section "Cfront 2.1/3.0 Compatibility Mode" on page 5-8.

- A non-const member function may be called for a const object. A warning is issued.
- A **const void *** value may be implicitly converted to a **void *** value (e.g., when passed as an argument).
- When, in determining the level of argument match for overloading, a reference parameter is initialized from an argument that requires a non-class standard conversion, the conversion counts as a user-defined conversion. (This is an outright bug, which unfortunately happens to be exploited in the NIH class libraries.)
- A reference to a non-const type may be initialized from a value that is a const-qualified version of the same type, but only if the value is the result of selecting a member from a const class object or a pointer to such an object.

- The cfront 2.1 transitional model for nested type support is simulated. In the transitional model a nested type is promoted to the file scope unless a type of the same name already exists at the file scope. It is an error to have two nested classes of the same name that need to be promoted to file scope or to define a type at file scope after the declaration of a nested class of the same name. This feature actually restricts the source language accepted by the compiler. This is necessary because of the affect this feature has on the name mangling of functions that use nested types in their signature. This feature does not apply to template classes.

- A cast to an array type is allowed. It is treated like a cast to a pointer to the array element type. A warning is issued.

- When an array is selected from a class, the type qualifiers on the class object (if any) are not preserved in the selected array. (In the normal mode, any type qualifiers on the object are preserved in the element type of the resultant array.)

- An identifier in a function is allowed to have the same name as a parameter of the function. A warning is issued.

- A value may be supplied on the return statement in a function with a void return type. A warning is issued.

- The destructor of a derived class may implicitly call the private destructor of a base class. In default mode this is an error, but in cfront mode it is reduced to a warning. For example:

```
class A          A(); ; class B : public A          B();
; B:: B()     // Error except in cfront mode
```

  Cfront 2.1 has a bug that causes a global identifier to be found when a member of a class or one of its base classes should actually be found. This bug is emulated in cfront compatibility. A warning is issued when, because of this feature, a nonstandard lookup is performed.

  The following conditions must be satisfied for the nonstandard lookup to be performed:

  - A member in a base class must have the same name as an identifier at the global scope. The member may be a function, static data member, or nonstatic data member. Member type names don't apply because a nested type will be promoted to the global scope by cfront, which disallows a later declaration of a type with the same name at the global scope.

  - The declaration of the global scope name must occur between the declaration of the derived class and the declaration of an out-of-line constructor or destructor. The global scope name must be a type name.

- No other member function definition---even one for an unrelated class---may appear between the destructor and the offending reference. This has the effect that the nonstandard lookup applies to only one class at any given point in time.

```
struct B {
     void func(const char*);
};
struct D : public B {
  public:
    D();
    void Init(const char* );
};
struct func  {
  func( const char* msg);
} ;

D::D(){}
void D::Init(const char* t)
{
  // Should call B::func --  calls func::func instead.
  new func(t);
}
```

- The global scope name must be present in a base class (B::func in this example) for the nonstandard lookup to occur. Even if the derived class were to have a member named func, it is still the presence of B::func that determines how the lookup will be performed.

- A parameter of type **const void *** is allowed on operator delete; it is treated as equivalent to **void ***. (This is permitted by cfront 2.1 but is an error in cfront 3.0.)

- A period (.) may be used for qualification where **::** should be used. Only **::** may be used as a global qualifier. Except for the global qualifier, the two kinds of qualifier operators may not be mixed in a given name (i.e., you may say A::B::C or A.B.C but not A::B.C or A.B::C). A period may not be used in a vacuous destructor reference nor in a qualifier that follows a template reference such as A<T>::B.

- Cfront 2.1 does not correctly look up names in friend functions that are inside class definitions. In this example function f should refer to the functions and variables (e.g., f1 and a1) from the class declaration. Instead, the global definitions are used.

```
int a1; int e1; void f1(); class A    int a1;   void f1();
friend void f()       int i1 = a1;  // cfront uses global a1
f1();  // cfront uses global f1    ;
```

Only the innermost class scope is (incorrectly) skipped by cfront as illustrated in the following example.

```
int a1;
int b1;
struct A {
   static int a1;
   class B {
     static int b1;
     friend void f()
     {
        int i1 = a1;  // cfront uses A::a1
        int j1 = b1;  // cfront uses global b1
     }
    };
};
```

Cfront 2.1/3.0 Compatibility Mode

The following extensions are accepted in cfront 2.1/3.0 compatibility mode.

- Because cfront does not check the accessibility of types, access errors for types are issued as warnings instead of errors.
- When matching arguments of an overloaded function, a const variable with value zero is not considered to be a null pointer constant.
- A reference to a pointer type may be initialized from a pointer value without use of a temporary even when the reference pointer type has additional type qualifiers above those present in the pointer value. For example:

```
int *p;
const int *&r = p;  // No temporary used
```

- An extra comma is allowed after the last argument in an argument list.

```
f(1, 2, );
```

- 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`.

Extensions

This section lists the language extensions supported by the Paragon system C++ compiler.

1. The **#pragma** [*tokens*] ANSI directive is supported. Any pragma that is not recognized is ignored.

2. The **#elif** *expression* ANSI directive is supported. This directive is like a combination of the **#else** and **#if** directives.

3. The following preprocessor macros are predefined (in addition to the ANSI-standard predefined macros **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, **__cplusplus**, and **__STDC__**):

   - **__i860**
   - **__i860__**
   - **__PARAGON__**
   - **__OSF1__**
   - **__PGC__**
   - **_PGC_**
   - **_COFF**
   - **__NODE** (only defined when compiling with **-nx**)

- **unix**
- **MACH**
- **CMU**

Note that some of these macro names begin and/or end with *two* underscores.

4. The **#ident** directive is supported. The syntax is:

   `#ident` `"`*string*`"`

   For certain assemblers, this results in a **.ident** directive being added to the output file.

Implementation-Defined Behavior

The sizes and alignments of the various C data types are shown in Table 5-1.:

Table 5-1. Sizes and Alignments of Data Types

Type	Size	Alignment
char	1 byte	byte
short	2 bytes	2-byte
int	4 bytes	4-byte
long int	4 bytes	4-byte
float	4 bytes	4-byte
double	8 bytes	8-byte
long double	8 bytes	8-byte
struct	(varies)	Alignment of field with largest alignment
union	(varies)	Alignment of member with largest alignment
array of *type*	*n* * size of *type*	Alignment of *type*

The search rules for **#include** directives are:

- If the pathname is enclosed in angle brackets, the compiler first searches the directories specified with the **-I** command line switch in the order specified, then the system include directory.
- If the pathname is enclosed in double quotes, the compiler first searches the current directory, then follows the search rules above.

Data Types

This section describes the scalar and aggregate data types recognized by the compiler, the format and alignment of each type in memory, and the range of values each type can take.

Scalars

A scalar data type (fundamental type) holds a single value, such as the integer value 42 or the bit string 10011. Table 5-2 lists scalar data types, their size and format.

Table 5-2. Scalar Data Types

Data Type	Size (bytes)	Format	Range
unsigned char	1	ordinal	0 to 255
[signed] char	1	two's complement integer	-128 to 127
unsigned short	2	ordinal	0 to 65535
[signed] short	2	two's complement integer	-32768 to 32767
unsigned int	4	ordinal	0 to 2^{32}-1
[signed] int	4	two's complement integer	-2^{31} to 2^{31}-1
unsigned long int	4	ordinal	0 to 2^{32}-1
[signed] long int	4	two's complement integer	-2^{31} to 2^{31}-1
float	4	ieee single-precision floating-point	10^{-37} to 10^{38}
double	8	ieee double-precision floating-point	10^{-307} to 10^{308}
long double	8	ieee double-precision floating-point	10^{-307} to 10^{308}
bit field	1 to 32 bits	ordinal	0 to 2^{size}-1
bit field	1 to 32 bits	two's complement integer	-2^{size-1} to 2^{size-1}-1
pointer	4	address	0 to 2^{32}-1
enum	4	two's complement integer	-2^{31} to 2^{31}-1

Table 5-3 lists the approximate ranges of floating-point data types.

Table 5-3. Floating-Point Data Type Ranges

Data Type	Binary Range	Decimal Range (approximate)	Decimal Digits of Precision (approximate)
float	2^{-126} to 2^{128}	10^{-37} to 10^{38}	7-8
double	2^{-1022} to 2^{1024}	10^{-307} to 10^{308}	15-16
long double	2^{-1022} to 2^{1024}	10^{-307} to 10^{308}	15-16

Alignment of Scalars

The alignment of a scalar data type is equal to its size. The following scalar alignments apply to individual scalars and to scalars that are elements of an array or members of a class, structure, or union.

char Aligned on a 1-byte boundary.

short Aligned on a 2-byte boundary.

[long] int Aligned on a 4-byte boundary.

enum Aligned on a 4-byte boundary.

pointer Aligned on a 4-byte boundary.

float Aligned on a 4-byte boundary.

double Aligned on a 8-byte boundary.

long double Aligned on a 8-byte boundary.

Wide characters are supported (character constants prefixed with an L of type wchar_t). The size of each wide character is 4 bytes.

Aggregate Data Types

An aggregate data type consists of one or more scalar data type objects. You can declare the following aggregate data types:

array One or more elements of a single data type placed in contiguous locations from first to last.

class A class that defines an object and its member functions. The object can contain fundamental data types or other aggregates including other classes. The class members are allocated in the order they appear in the definition but may not occupy contiguous locations.

struct A structure that can contain different data types. The members are allocated in the order they appear in the definition but may not occupy contiguous locations. When a struct is defined with member functions, its alignment issues are the same as those for a class.

union A single location that can contain any of a specified set of scalar or aggregate data types. A union can have only one value at a time. The data type of the union member to which data is assigned determines the data type of the union after that assignment.

Class and Object Data Layout

Class and structure objects with no virtual entities and with no base classes (just direct data field members) are laid out in the same manner as C structures. The following section describes the alignment and size of these C-like structures. C++ classes (and structures as a special case of a class) are more difficult to describe. Their alignment and size is determined by compiler-generated fields in addition to user–specified fields. The following paragraphs describe how storage is laid out for more general classes. The user is warned that the alignment and size of a class (or structure) is dependent on the existence and placement of direct and virtual base classes and of virtual function information. The information that follows is for informational purposes only and reflects the current implementation. Note that it is subject to change, and only and reflects the current implementation. Do not make assumptions about the layout of complex classes or structures.

All classes are laid out in the same general way, using the following pattern (in the sequence indicated):

- First, storage for all of the direct base classes (which implicitly includes storage for nonvirtual indirect base classes as well):
  - When the direct base class is also virtual, only enough space is set aside for a pointer to the actual storage, which appears later.
  - In the case of a nonvirtual direct base class, enough storage is set aside for its own nonvirtual base classes, its virtual base class pointers, its own fields, and its virtual function information, but no space is allocated for its virtual base classes.
- Next, storage for the class's own fields.
- Next, storage for virtual function information (typically, a pointer to a virtual function table).
- Finally, storage for its virtual base classes, with space enough in each case for its own nonvirtual base classes, virtual base class pointers, fields, and virtual function information.

Structure Alignment

This section uses structure to mean a C-like structure, one with only direct data field members. In C++ these structures may be declared as **struct** or **class**. The layout is not affected by the visibility, public or private, of the data members.

The alignment of a structure or union affects how much space the structure occupies and how efficiently the processor can address the structure members. The user has some control over alignment and can specify *natural alignment* or *quad alignment* for structures:

Natural alignment	The default alignment. Structures are aligned on the boundary of their most-strictly-aligned member.
Quad alignment	Aligns 16-byte or larger top-level objects, such as local arrays and structures, on quad-word boundaries. You specify quad alignment using the **–Mquad** option. For more information on the **–Mquad** option, see Chapter 2. Quad alignment has no effect on structures which are members of other structures or on structures that are members of arrays.

Structure alignment can result in unused space, called *padding*, between members of the structure and between the last member and the end of the space occupied by the structure. The padding at the end of the structure is called *tail padding*. Because of differences in padding under different alignments, changing the alignment can change the tail padding.

The rules for structure member alignment are:

Scalar types — Align according to their natural architectural alignment. For example, a data type aligns on a 2-byte boundary.

Array types — Align according to the alignment of the array elements. For example, an array of short data type aligns on a 2-byte boundary.

class / structure / union types

Align according to the most restrictive alignment of the enclosing members. For example the union **un1** below aligns on a 4-byte boundary since the alignment of c, the most restrictive element, is four:

```
union un1 {
  short a;  /* 2 bytes */
  char  b;  /* 1 byte  */
  int   c;  /* 4 bytes */
};
```

By specifying quad alignment with the **–Mquad** option, a structure whose size is at least 16 bytes may use different tail padding from a structure using natural alignment (depending on the contents of the structure). This change represents the difference between word-level tail padding and quad-word level tail padding. Note that natural alignment and quad alignment structures differ only in tail padding. Figures 5-1 and 5-2 illustrate natural and quad alignment. Consider the following structure:

```
struct strc1 {
  char  a;  /* occupies byte 0             */
  short b;  /* occupies bytes 2 and 3      */
  char  c;  /* occupies byte 4             */
  int   d;  /* occupies bytes 8 through 11 */
};
```

Figure 5-1 shows that with natural alignment, the size allocated is 12 bytes. The memory allocated for this structure uses no padding for either natural or quad alignment, since the structure is less than 16 bytes (a quad-word).

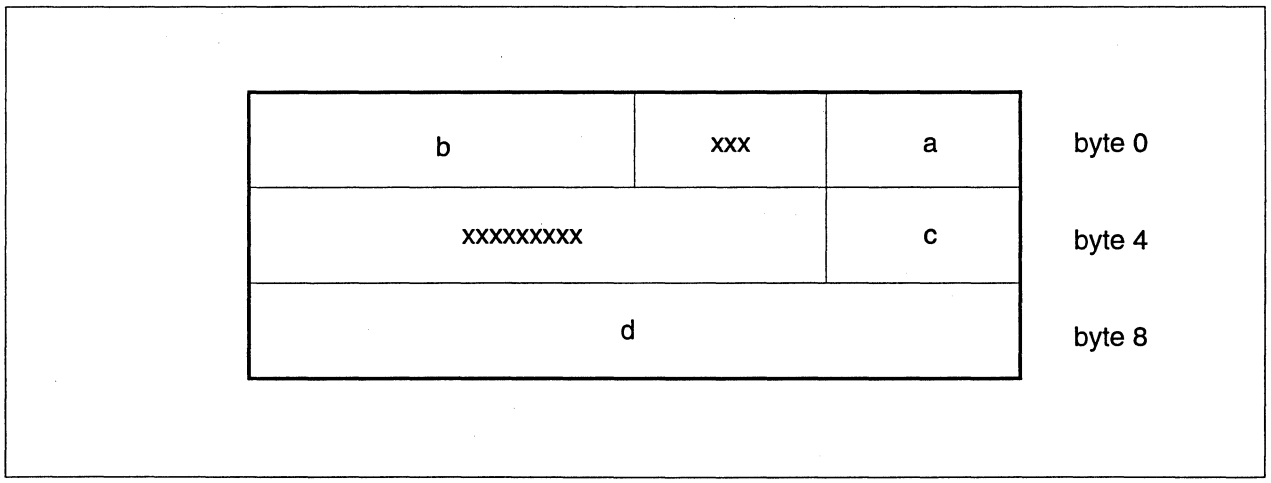


Figure 5-1. Natural Alignment

Figure 5-2 shows how a structure with more than 16 bytes occupies memory using quad alignment. This structure shows how quad alignment changes the tail padding and causes the structure to align on a quad-word boundary. In this case quad alignment increases the amount of tail padding relative to natural alignment.

Specifying quad alignment for *strc2* aligns the structure as shown in Figure 5-2. Using natural alignment this structure's second element would align according to the restriction of the largest element, an integer, and the structure would be padded to byte 20 instead of to byte 32 as for quad alignment.

```
struct strc2{
  int   m1[4];  /* occupies bytes 0 through 15 */
  short m2;     /* occupies bytes 16 and 17    */
} st;
```

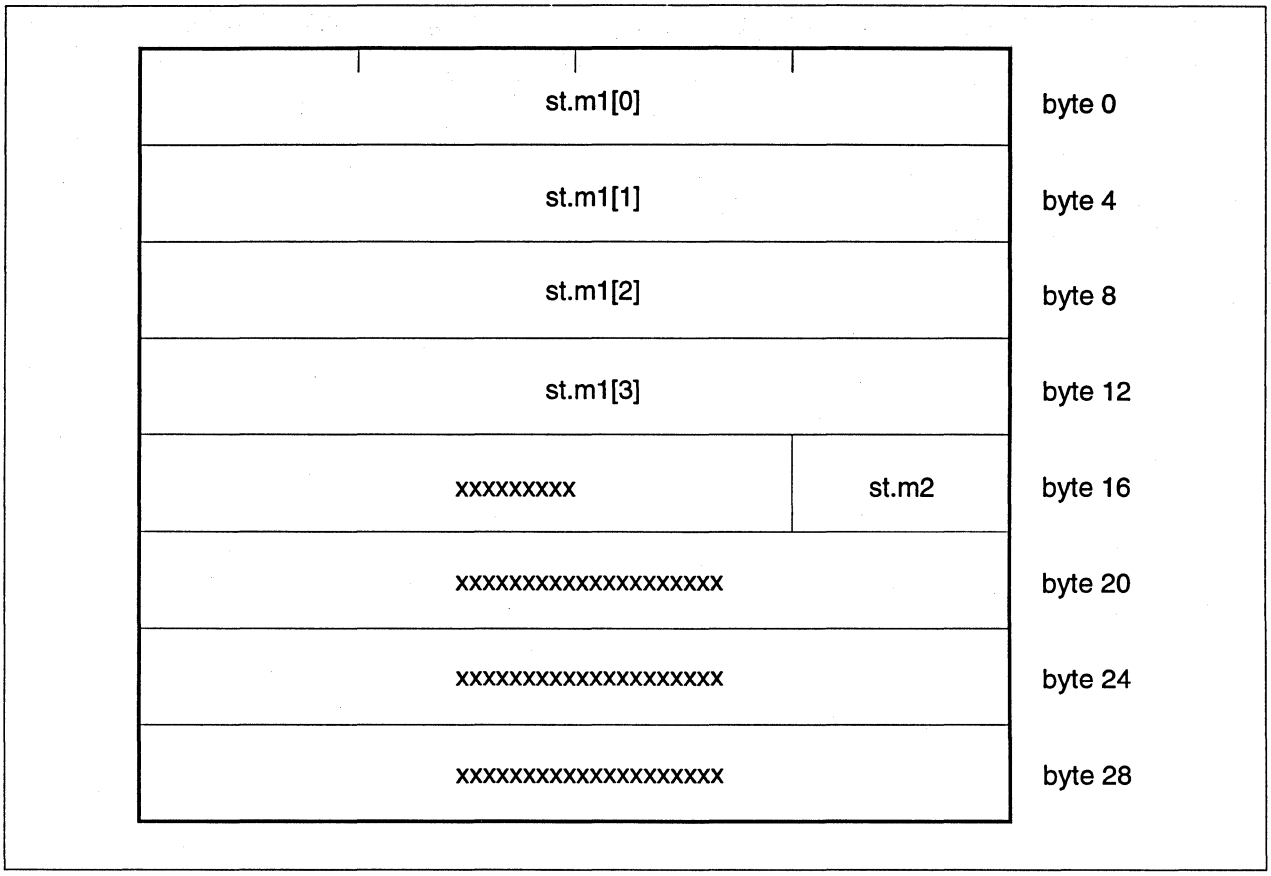


Figure 5-2. Quad Alignment

Bit-field Alignment

Every bit field lies entirely within some bit-field container that has the same size and alignment as its declared type. The size and alignment required by a container are defined by its declared type. For example, if the bit field is of type **int**, the container alignment is four bytes. A bit field can cross byte boundaries but cannot cross a container boundary (if the container consists of greater than one byte).

The size of a bit field cannot exceed the size of its container. The compiler aligns an individual bit field when the bit field, unaligned, would overrun the end of the container in which it starts. A bit-field size of zero also forces bit-field alignment. You can determine the alignment of a bit field and the position of the bit field within a structure as follows:

- The byte position of a bit field within a container is the current byte offset in the structure modulo the container alignment. This value is the byte offset relative to the most recent container alignment boundary. For example, if the container alignment is 1, the byte position is always 0. If the container alignment is 4, the byte position can be 0, 1, 2, or 3.

- The bit position of the bit field is the number of bits already allocated in the current byte, plus 8 times the container byte position. This value is the bit offset, in the range 0 to *n* relative to the most recent container alignment boundary (*n* is 8 times the number of bytes in the container).

- If the value of the container bit position plus the size in bits of the new bit field is greater than the size of the container or if the size of the new bit field is zero, the compiler aligns the bit field on the next container alignment boundary. Bit-field alignment can result in padding of up to 32 bits. If the bit-field size is nonzero and the bit field fits entirely within the current container, the compiler does not align the bit field.

Other Type Keywords

The **void** data type is neither a scalar nor an aggregate. You can use **void** or **void*** as the return type of a function to indicate the function does not return a value, or as a pointer to an unspecified data type, respectively.

The **const** and **volatile** type qualifiers do not in themselves define data types, but associate attributes with other types. Use **const** to specify that an identifier is a constant and is not to be changed. Use **volatile** to prevent optimization problems with data that can be changed from outside the program, such as memory-mapped I/O buffers.

Libraries 6

This chapter describes the libraries included with the Paragon™ system C++ Compiler. The libraries include the following:

- Compiler libraries.
- Iostream libraries.
- Complex arithmetic library.

Compiler Libraries

This section introduces you to the compiler libraries and covers the following topics:

- The format of library functions.
- How to link the math libraries during compilation.
- How to include the math header file.

The following startup routines, function libraries, object files and source files are provided with the Paragon system C++ compiler or C compiler:

crt0.o	A startup routine for the compilation environment.
crt0.s	The assembly-language source of the startup routine *crt0.o*.
libb.a	A math library that provides more accurate (approximately one-half the error), but slower, math routines. This library contains both single-precision and double-precision versions of the math functions listed in the ANSI C standard.

libc.a	The standard C run-time library. This library contains all of the run-time functions listed in the ANSI C standard.
libC.a	The C++ run-time library. This library contains all of the run-time functions for C++.
libstrm.a	Includes all the functions for C++ basic input and output with iostream.
libcmplx.a	Includes the type complex with constructors and functions for working with complex numbers.
libm.a	A math library that provides faster (approximately twice as fast), but less accurate, math routines. This library contains both single-precision and double-precision versions of the math functions listed in the ANSI C standard.
libic.a	The compiler support library. All routines compiled with the compiler must include this library. The functions in this library perform operations such as conversion between unsigned and floating-point, IEEE-conformant floating-point division, and integer division.
subchk.o	The object file containing routines that handle bounds checking. The optional bounds checking for C programs are enabled with the **–Mbounds** option on the **icc** command line.

The standard C++ library routines are listed in this chapter. For more information on the standard C run-time library routines, refer to the ANSI C standard. For more information about the libraries provided with the Paragon system C compiler, refer to the *Paragon™ System C Compiler User's Guide*.

Linking to the Math Library

The names of the math library files are *libb.a* (Berkeley) and *libm.a.* The math library is not linked in by default. If you want to use a math library, link in the math library after your source file, as in the following example. This example links in the *libm.a* library before the *libmylib.a* library.

```
$ icc myprog.c -lm -lmylib
```

Including the Math Header File

The name of the math header file is *math.h*. Include the math header file in all of your source files that use a math library routine as in the following example, which calculates the inverse cosine of $\pi/3$.

```
#include <math.h>

#define PI 3.1415926535

main()
{
   double x, y;
   x = PI/3.0;
   y = acos(x);
}
```

Math Functions

Table 6-1 provides a cross-reference of math library functions. Use this table to find the function that satisfies the required operation and type of operands.

Table 6-1. Math Functions by Operation and Arguments (1 of 2)

Operation	Type of Arguments (Double)	Type of Arguments (Single)
absolute value	fabs	
ceiling	ceil	
floor	floor	
floating remainder	fmod	
fraction/exponent	frexp	
floating times power of 2	ldexp	
fraction/integer	modf	
power	pow	powf
square root	sqrt	sqrtf
cosine	cos	cosf
sine	sin	sinf
tangent	tan	tanf
arc cosine	acos	acosf

Table 6-1. Math Functions by Operation and Arguments (2 of 2)

Operation	Type of Arguments (Double)	Type of Arguments (Single)
arc sin	asin	asinf
arc tangent	atan	atanf
arc tangent of quotient	atan2	atan2f
hyperbolic cosine	cosh	coshf
hyperbolic sine	sinh	sinhf
hyperbolic tangent	tanh	tanhf
exponential	exp	expf
natural logarithm	log	logf
log base 10	log10	log10f

Standard C Library Calls

This section lists the standard C library calls supplied with the system library, *libc.a.*

abort	fputc	printf	strncat
abs	pfuts	puts	strncmp
atexit	fread	qsort	strncpy
atoi	fseek	raise	strpbrk
atol	fsetpos	rand	strrchr
bsearch	ftell	remove	strspn
clearerr	fwrite	rename	strstr
clock	getenv	rewind	strtok
difftime	gets	scanf	strtol
div	gmtime	setvbuf	strtoul
exit	labs	strcat	strxfrm
fclose	ldiv	strchr	system
feof	malloc	strcmp	tmpfile
ferror	memchr	strcoll	tmpnam
fflush	memcmp	strcpy	ungetc
fgetc	memcpy	strcspn	vfprintf
fgetpos	memmove	strerror	vfscanf
fgets	memset	strftime	
fopen	perror	strlen	

Non-Standard C Library Calls

This section lists the C library calls that are not standard, but are supplied with the system library, *libc.a.*

assert	getcwd
bcopy	getopt
bzero	isatty
drand48	locale
execl	mktemp
execle	setjmp
fdopen	sleep

Setting the SBRK Size

There are C user-callable routines that set the sbrk size. The routine **__malloc_sbsz(*x*)**, where *x* is an integer value, first rounds *x* to a multiple of 16 and sets the sbrk size to this (rounded) value. The following is an example.

```
int x = 256;
__malloc_sbsz(x); /* sets sbrk size to 256 */
```

The Iostream Library

Iostream is the standard C++ input and output library. The iostream package allows operations on streams of characters. The iostream package is implemented in a layered two-level approach. At the lowest layer, implemented by the streambuf class, operations are defined for buffering the input and output of sequences of characters (the terms input and output are used in a general sense in relation to a sequence of characters). Higher layers in iostream add formatting and/or file specific operations to this lowest layer.

Iostreams Header Files

The iostream package consists of a number of classes and routines defined in the following header files:

fstream.h	File handling input output routines.
iomanip.h	Iostream manipulators for insert and extract special features.
iostream.h	Defines for basic input and output operations.
stdiostream.h	Specialized routines for input and output through **FILE** structs.

stream.h	Includes all the iostream .h files and also some additional compatibility definitions for older versions of C++.
strstream.h	Array handling input output routines.

Iostreams Usage and Class Hierarchy

The iostreams package for input and output contains a number of classes including the base class ios. The ios class contains state information supporting formatting operations on streams; the input and output classes including the istream and ostream classes as well as additional file and string support classes are also part of iostream. After presenting an example showing iostream usage, the following sections define the structure of each of the iostream classes.

Using iostreams

The following example uses the predefined output stream cout (which is connected to stdout.) It also uses the overloaded operator << from class ostream to output the string "hello world\n" to the terminal:

```
#include <iostream.h>
void main(void)
{
 cout << "hello world\n";
}
```

The following example reads in an integer from stdin and then outputs it.

```
#include <iostream.h>
void main(void)
{
int weight;
cout << "Enter your weight:" << flush;
cin >> weight;
cout << endl << "you weigh " << weight << endl;
}
```

The following example concatenates the files *first.txt* and *second.txt* into file *output.txt*.

```
#include <fstream.h>
void main(void)
{
 ifstream infile1("first.txt");
 if (!infile1)
    cerr << "unable to open file first.txt\n";
 ifstream infile2("second.txt");
 if (!infile2)
    cerr << "unable to open file second.txt\n";
 ofstream outfile("outfile.txt");
 if (!outfile)
    cerr << "unable to open file output.txt\n";
 while (infile1.getc(c))
    outfile.putc(c);
 while (infile2.getc(c))
    outfile.putc(c);
}
```

The other classes in iostream use the streambuf class, or a derived class, to support buffering of streams. They add state information and functions to add formatted/unformatted file/incore input and/or output to the streambuf-derived buffering classes.

The following is a list of predefined streams:

cin	Standard input stream (uses file descriptor 0).
cout	Standard output stream (uses file descriptor 1).
cerr	Standard error (unbuffered) stream (uses file descriptor 2).
clog	Standard error (buffered) stream (uses file descriptor 2).

Class ios

The ios class is the base class for iostream operations. This class defines a number of public functions for streams. It also defines *enums* used to specify formatting state, error state, open mode, seek directions that control input and output operations. The following section lists enumerated types defined in the file *iostream.h*.

Class ios Enumerated Types

io_state enumc

```
enum io_state  {
   goodbit=0,
   eofbit=1,
   failbit=2,
   badbit=4,
   hardfail=0200
};
```

open_mode enum

```
enum open_mode {
   in=1,
   out=2,
   ate=4,
   app=010,
   trunc=020,
   nocreate=040,
   noreplace=0100
} ;
```

seek_dir enum

```
enum seek_dir  {
   beg=0,
   cur=1,
   end=2
};
format flags enum

enum {                          /* flags for controlling format */
   skipws=01,                       /* skip whitespace on input */
   left=02,
   right=04,
   internal=010,                          /* padding location */
   dec=020,
   oct=040,
   hex=0100,                               /* conversion base */
   showbase=0200,
```

```
    showpoint=0400,                              /* modifiers */
    uppercase=01000,
    showpos=02000,                               /* modifiers */
    scientific=04000,
    fixed=010000,                  /* floating point notation */
    unitbuf=020000,
    stdio=040000                 /* stuff to control flushing */
} ;
```

Class ios Constructor and Destructor

ios

The ios constructor routine is defined as follows:

ios(streambuf *s);

This creates an ios object to represent a stream's state and associates streambuf *s* with the stream.

~ios

The ios destructor routine is defined as follows:

virtual ~ios();

Class ios – Public Functions

This sections lists the public member functions defined for the class ios.

Note

Extension functions allow derived classes to add additional format flags or state information.

bad

Error status function.

int bad();

The routine returns a non-zero value if the **ios_state** *badbit* is set (indicating a severe **rdbuf()** failure). The routine returns 0 otherwise.

bitalloc

Extension function.

static long bitalloc();

The routine returns a long with a single bit set. This is a newly allocated bit that the derived class may use and pass to **setf()**.

clear

Error status function.

void clear(int *state* **= 0);**

The argument *state* is used as new error state and sets the value (mask) for the ios enum **ios_state**. If *state* is zero, all bits of **ios_state** are cleared.

eof

Error status function.

int eof();

Returns not-zero if the **ios_state** enum *eofbit* is set (indicating an end-of-file has been reached) ; 0 (zero) otherwise.

fail

Error status function.

int fail();

Returns non-zero if the **ios_state** enum *badbit* or *failbit* is set (indicating a get or conversion failure, stream may be usable); 0 (zero) otherwise.

fill

Format access function.

int fill();
int fill(char *c1***);**

Return the stream's fill character. With an argument, set the fill character to *c1*

flags

Format access function. See the format flags enum above in this section for details on the various flags.

long flags();
long flags(long *flag*);

Return the format flags. With an argument, set the format flags.

good

Error status function.

int good();

Returns non-zero if no error bits are set; 0 (zero) otherwise.

iword

Extension function.

long& iword(int *index*);

The parameter *index* should have been allocated by a call to **xalloc()** Returns a reference to a user-defined word at *index* offset into an array managed by **xalloc()**.

precision

Format access function.

int precision();
int precision(int *fpvp*);

Return the precision used to format floating-point values. With an argument, set the precision used to format floating-point values.

pword

Extension function.

void* & pword(int *index*);

The parameter *index* should have been allocated by a call to **xalloc()**. Returns a reference to a user-defined word at *index* offset into an array managed by **xalloc()**.

rdbuf

streambuf* rdbuf();

Returns a pointer to the associated streambuf used for buffering.

rdstate

Error status function.

int rdstate();

Returns the ios error state flags. See the enum **io_state**.

setf

Format access function.

long setf(long *setbits***);**

Set the format flag bits specified in *setbits*. Returns the previous settings.

long setf(long *setbits***, long** *field***);**

Format state function. Set the format flag bits identified by *field* to the bits specified in *setbits*.

skip

Old stream package compatibility routine.

int skip(int *i***);**

sync_with_stdio

static void sync_with_stdio();

Allows mixing stdio and iostreams. I/0 will be flushed appropriately so it appears in the expected order.

tie

ostream* tie();
ostream* tie(ostream* *p*);

Returns the tie variable. With an argument, sets the *tie* variable to *p*.

The *tie* variable is used to force flushing of one stream before a get or put operation is performed by another stream. If the tie variable is not null, and more character must be got or put, the ostream pointed to by *p* is flushed.

unsetf

Format access function.

long unsetf(long *setbits*);

Unset the format flag bits specified in *setbits*. Returns the previous settings. Refer to the format flags enum for a description of the various format flags.

width

Format access function.

int width();
int width(int *wval*);

Return the field width used to pad out, with the fill character, values being inserted. With an argument, *wval*, sets the field width used to pad out (with the fill character) values being inserted.

xalloc

Extension function.

static int xalloc();

Returns a new index into an array of words. Used to extend formatting or other state information. Word is accessed by **iword()** or **pword()** functions.

Operators

Error status operator functions.

int operator !();

Returns non-zero if *badbit* or *failbit* is set.

int operator void*();

Returns 0 (zero) if *badbit* or *failbit* is set. Returns non-zero otherwise.

Class istream

The class istream supports operations to fetch and to format sequences of characters inserted from streambufs. The istream functions are included with *iostream.h*.

Class istream constructor and destructor

istream

The istream constructor.

istream(streambuf* *s*);

Associate streambuf pointed to by *s* to stream.

~istream

The istream destructor.

virtual ~istream();

Class istream – Public Functions

gcount

Input routine.

int gcount();

Returns the number of characters extracted by the last unformatted **get()**, **getline()**, or **read()**. Formatting functions may call these functions and change this number.

get

istream& get(char* *ptr*, **int** *len*, **char** *delimiter*='\n');

Unformatted input function. Extracts *len* bytes and puts them starting at *ptr*. Extraction ends if a delimiter is found. The delimiter is not copied. A terminating NULL is always added.

istream& get(unsigned char* *ptr*, **int** *len*,
char *delimiter*='\n');

Unformatted input function. Extracts *len* bytes and puts them starting at *ptr*. Extraction ends if a delimiter is found. The delimiter is not copied. A terminating NULL is always added.

istream& get(char& *c*);

Unformatted input function. Extracts a single character and stores it in *c*.

istream& get(unsigned char& *c*);

Unformatted input function. Extracts a single character and stores it in *c*.

istream& get(streambuf& *s*, **char** *delimiter*='\n');

Unformatted input function. Extracts characters and stores them in streambuf *s*. Stops if end-of-file or the store fails, or if a delimiter is found.

int get();

Unformatted input function. Extracts and returns the extracted character. end-of-file is returned for end-of-file.

getline

istream& getline(char * *ptr*, **int** *len*,
char *delimiter*='\n');

Unformatted input function. Operates like **get()** with similar parameter list, but the delimiter is copied.

istream& getline(unsigned char * *ptr*, **int** *len*,
char *delimiter*='\n');

Unformatted input function. Operates like **get()** with similar parameter list, but the delimiter is copied.

ignore

istream& ignore(int *len* **= 1, int** *delimiter*=**EOF);**

Unformatted input function. Extract but do not store *len* characters or until delimiter is extracted.

ipfx

int ipfx(int *need*=**0);**

Input prefix function. Checks state of input stream. If error state is non-zero returns zero. Otherwise may flush tied iostream, skip whitespace, etc.

Formatted input functions call **ipfx(1)**, unformatted input functions call **ipfx(0)**.

peek

int peek();

Returns the next character without extracting it if **!ipfx(1)** or if at end-of-file returns end-of-file; Otherwise it returns the next character.

putback

istream& putback(char *c*);

Attempts to back up streambuf pointer. *c* is last character extracted.

read

istream& read(char* *str*, **int** *n*);

Unformatted input function. Extracts *n* characters and stores them at *str*. If the end-of-file is hit, read copies up to it, sets *failbit*, and the count of bytes read will be returned if **gcount()** is called.

istream& read(unsigned char* *ustr*, **int** *n*);

Unformatted input function. Extracts *n* characters and stores them at *str*. If end-of-file is hit, read copies up to it, sets *failbit*, and the count of bytes read will be returned if **gcount()** is called.

seekg

istream& seekg(streampos *pos*);

Repositions the streambufs *get* pointer. See class streambuf **seekpos()**.

istream& seekg(streamoff *val1*, **seek_dir** *direction*);

Repositions the streambufs *get* pointer. See class streambuf **seekoff()**.

sync

int sync();

Synchronizes internal data and state with streambuf external source.

tellg

streampos tellg();

Returns the current position of streambuf's get pointer.

Class ostream

Supports operations to possibly format and to store sequences of characters inserted into streambufs. The ostream functions are included with *iostream.h*.

Class istream constructor and destructor

ostream

The ostream constructor.

ostream(streambuf* *s*);

Associate streambuf pointed to by *s* to stream.

~ostream

The istream destructor.

virtual ~ostream();

Class ostream – Public Functions

flush

ostream& flush();

Flushes the buffer associated with streambuf.

opfx

int opfx();

Output prefix function. Checks error state, returns zero if error. Flushes a *tied* stream.

osfx

void osfx();

Output suffix function. Called at end of inserter routines to cleanup by flushing if necessary.

put

ostream& put(char *c***);**

Unformatted output function. Puts char *c* into the associated streambuf.

seekp

ostream& seekp(streampos *vpos***);**

Repositions the streambufs *put* pointer to *vpos*. See class streambuf **seekpos**().

ostream& seekp(streamoff *pos***, seek_dir** *dir***);**

Repositions the streambufs *put* pointer. See class streambuf **seekoff**().

tellp

streampos tellp();

Returns the current position of the associated streambuf's put pointer. See class streambuf **seekoff()**.

write

ostream& write(const char* *ptr*, **int** *n*);

Unformatted output function. Writes *n* characters starting at *ptr* into the associated streambuf.

ostream& write(const unsigned char* *ptr*, **int** *n*);

ostream Operators

ostream& operator << (char *arg*);

This function outputs *arg*. Also defined for *arg* of type **unsigned char**, **short**, **unsigned short**, **int**, **unsigned int**, **long**, **unsigned long**, **float**, **double**, **const char ***, **void ***, **streambuf ***, **ostream& (*)(ostream&)**, and **ios& (*)(ios&)**.

Class iostream

The iostream class combines istream and ostream abstractions. The following classes are derived from istream, ostream, or iostream. They are the high level classes that users are most apt to use for general file I/O. These classes are defined in the include file *fstream.h*.

Three classes support input and/or output formatted file I/0:

ifstream	Supports formatted buffered input file I/O.
ofstream	Supports formatted buffered output file I/O.
fstream	Supports formatted buffered input/output file I/O.

iostream Constructor and Destructor

iostream

The following constructor is defined for class iostream.

iostream(streambuf* *psb*);

~iostream

The iostream destructor.

virtual ~iostream();

fstream Constructor and Destructor

fstream

The following constructors are defined in class fstream.

fstream();

Create unopened fstream.

fstream(const char* *filename*, **int** *mode*,
int *protection*);

Create fstream; open *filename* with specified *mode* and specified *protection*. Default mode is input/output. Default protection is 0644 in UNIX terms.

fstream(int *filedescriptor*);

Create fstream and connect to open *filedescriptor*.

fstream(int *filedescriptor*, **char*** *p*, **int** *len*);

Create fstream and connect to open *filedescriptor*. Use buffer at *p* of length *len*. If *p* is 0 or *len* is 0 then filebuf is unbuffered.

~fstream

The fstream destructor.

virtual ~fstream();

ofstream Constructor and Destructor

ofstream

The following constructors are defined in class ofstream.

ofstream();

Create unopened ofstream.

ofstream(const char* *filename*, **int** *mode*,
int *protection*);

Create ofstream; open filename with mode and protection.

Default mode is output. Default protection is 0644 in UNIX terms.

ofstream(int *filedescriptor*);

Create ofstream and connect to open *filedescriptor.*

ofstream(int *filedescriptor*, **char *** *p*, **int** *len*);

Create ofstream and connect to open file descriptor. Use buffer at *p* of length *len*. If *p* is 0 or *len* is 0 then filebuf is unbuffered.

~oftream

The ofstream destructor.

virtual ~ofstream();

ifstream Constructor and Destructor

ifstream

The following constructors are defined in class ifstream.

ifstream();

Create unopened ifstream.

ifstream(const char* *filename*, **int** *mode*,
int *protection*);

Create ifstream; open *filename* with *mode* and *protection*.

Default mode is input. Default protection is 0644 in UNIX terms.

ifstream(int *filedescriptor*);

Create ifstream and connect to open filedescriptor.

ifstream(int *filedescriptor*, **char*** *p*, **int** *len*);

Create ifstream and connect to open *filedescriptor*. Use buffer at *p* of length *len*. if *p* is 0 or *len* is 0 then filebuf is unbuffered.

~ifstream

The ifstream destructor.

virtual ~ifstream();

Class fstream, ifstream, ofstream – Public functions

The following public member functions have been defined for all three classes fstream, ifstream, and ofstream.

attach

void attach(int *filedescriptor*);

Attaches the stream to *filedescriptor*. The *failbit* is set on error.

close

void close();

Closes the associated filebuf.

open

void open(char* *filename*, **int** *mode*, **int** *protection*);

Opens a file whose name is *filename* and attaches a stream to open file. Mode and protection defaults are similar to those in the constructor.

rdbuf

filebuf * rdbuf();

Returns pointer to the associated filebuf.

setbuf

void setbuf(char* *ptr*, **int** *len*);

Allow the use of *len* bytes starting at *ptr* as the buffer. If *ptr* is NULL or *len* is 0 (zero) subsequent operations are unbuffered.

Returns the pointer to streambuf or 0 (zero) if the request to use buffer or allow unbuffered operations can not be accepted.

Class ostream_withassign

ostream_withassign

The ostream_withassign constructor.

ostream_withassign() ;

~ostream_withassign

The ostream_withassign destructor.

virtual ~ostream_withassign();

Class iostream_withassign

iostream_withassign

The iostream_withassign constructor.

iostream_withassign() ;

~iostream_withassign

The iostream_withassign destructor.

virtual ~iostream_withassign();

Class istream_withassign

istream_withassign

The istream_withassign constructor.

istream_withassign() ;

~istream_withassign

The istream_withassign destructor.

virtual ~istream_withassign();

Class istrstream

This class supports formatted buffered input from incore character arrays. The istrstream functions are included with *strstream.h*.

Class istrstream Constructors and Destructor

istrstream

istrstream(char * *p*);

Make an istrstream object for string starting at *p* that is the length of the string *p*.

istrstream(char * *p*, int *len*);

Make an istrstream object for string starting at *p* that is the length of length *len*.

~istrstream

virtual ~istrstream();

The istrstream destructor.

Class istrstream – Public functions

rdbuf

strstreambuf* rdbuf();

Return the associated strstreambuf.

Class ostrstream

The ostrsstream class supports formatted buffered output to incore character arrays. The ostrstream functions are included with *strstream.h*.

Class ostrstream constructor and destructor

ostrstream

The following constructors are defined in class ostrstream:

ostrstream();

Dynamically allocate buffer to use for output buffer.

ostrstream(char* *p*, **int** *len*, **int** *mode*);

Use array of *len* characters beginning at *p* as output buffer.

A put pointer is normally set to *p*, but if `ios::ate` or `ios::app` are set in *mode*, then *p* is assumed to locate a null-terminated string and the put pointer will be initialized to the location of the null character.

~ostrstream

The ostrstream destructor.

virtual ~ostrstream();

Class ostrstream – Public Functions

The following public member functions are defined for ostrstream.

pcount

int pcount();

Returns the number of characters stored in the buffer.

rdbuf

strstreambuf* rdbuf();

Returns the associated strstreambuf.

str

char* str();

Returns a pointer to the first character of the current array.

Class strstream

The strstream class supports formatted buffered input/output to incore character arrays. The strstream functions are included with *strstream.h*.

Class strstream constructor and destructor

strstream

The following constructors are defined in class strstream:

strstream();

Dynamically allocate buffer.

strstream(char * *p*, **int** *len*, **int** *mode*)

Use array of *len* characters beginning at *p* as the input/output buffer.

~strstream

The strsttream destructor.

virtual ~strstream();

Class strstream – Public Functions

The following public member functions are defined:

str

char* str();

Returns a pointer to the first char of the current array.

Class stdiostream

The class stdiostream supports formatted buffered input/output file I/O in a manner similar to the stdio C library. The stdiostream functions are included with *stdiostream.h*.

Class stdiostream constructor and destructor

stdiostream

The following constructor is defined in class stdiostream:

stdiostream(FILE* *fp*);

Construct an empty file pointer *fp*.

~stdiostream

The stdiostream destructor.

virtual ~stdiostream();

Class streambuf

The streambuf class is a character buffering class. The streambuf class implements a character buffering package that supports a buffer, a sequence of characters, and *put* and/or *get* pointers into that sequence. The pointers will be used to support operations to put (insert) characters into that buffer and/or to get (extract) characters from that buffer. The source of the got characters and the sink for the put characters are dependent on the derived class of the streambuf class. The streambuf functions are included with *iostream.h*.

Class streambuf constructor and destructor

streambuf

The following constructors are defined in class streambuf:

streambuf();

Construct an empty buffer.

streambuf(char * *p*, **int** *len*)

Construct an empty buffer starting at *p* of length *len*.

~streambuf

The streambuf destructor.

virtual ~streambuf();

Class streambuf – Public Functions

The following public member functions are defined in class streambuf (defined in file *iostream.h*).

in_avail

int in_avail();

Returns the number of characters available in the get area.

out_waiting

int out_waiting();

Returns the number of characters still buffered in the put area.

sbumpc

int sbumpc();

Move the get pointer forward one; returns the character skipped or end-of-file if get pointer is at the end of the sequence.

setbuf

streambuf * setbuf(char* *ptr*, **int** *len*);

Allow the use of *len* bytes starting at *ptr* as the buffer. If *ptr* is NULL or *len* is 0 (zero) request unbuffered operations.

Returns pointer to streambuf or 0 (zero) if the request to use buffer or allow unbuffered operations can not be accepted.

seekoff

streampos seekoff(streamoff, seek_dir,
int *mode*=**ios::in|ios::out);**

Reposition the put and/or get pointers. The parameter *streamoff* is taken as a signed byte offset. The parameter *seekdir* is one of **ios::beg** (beginning of the stream), **ios::cur** (current position), or **ios::end** (end of the stream). The mode parameter mode is one or both bits **ios::in** (apply to the get pointer), and **ios::out** (apply to the put pointer).

Returns end-of-file if the class doesn't support repositioning or on error. Returns new position otherwise.

seekpos

streampos seekpos(streampos,int *mode*= **ios::in|ios::out);**

Reposition the put and/or get pointer to streampos. Mode specifies get or put pointers as in **seekoff()**.

sgetc

int sgetc();

Returns the character after the get pointer. The get pointer is not changed. Returns end-of-file if no character is available.

sgetn

int sgetn(char* *ptr*, **int** *n*);

Copy *n* characters following the get pointer to location starting at *ptr*. The get pointer is repositioned after the fetched characters.

Returns the number of characters actually copied (which may be less than *n*.)

snextc

int snextc();

Move the get pointer ahead one character and return the character following the new position.

Returns end-of-file if at the end of the sequence currently, or after the get pointer has been advanced.

sputbackc

int sputbackc(*char*);

Move the get pointer back one character. The parameter *char* must represent the character following the get pointer. Conceptually the *char* is put back into the get buffer.

Returns end-of-file on error.

sputc

int sputc(*char*);

Put *char* following the put character and advance the put pointer beyond the stored character (possibly extending the sequence).

Returns end-of-file on error.

sputn

int sputn(const char* *ptr*, int *n*);

Copy n chars starting at *ptr* to the put area following the put pointer and advance the put pointer past the added characters.

Returns the number of characters stored (which may be less than *n*).

stossc

void stossc();

Move the get pointer forward one character. If at end of sequence, do nothing.

sync

virtual int sync();

Derived class dependent, but typically used to synchronize the streambuf class with the source (or sink) of the sequence of characters being buffered.

Returns end-of-file on error.

The following three classes are derived from class streambuf.

Class filebuf

The filebuf class is derived from streambuf and adds support for I/O through file descriptors. Operations for opening, closing and seeking in files are added.

Class filebuf Constructors and Destructors

filebuf

The following constructors are defined in class filebuf:

filebuf();

Construct an empty filebuf.

filebuf(int *filedescriptor*);

Construct an empty filebuf attached to *filedescriptor.*

filebuf(int *filedescriptor*, **char*** *p* , **int** *len*);

Construct an empty filebuf attached to *filedescriptor* with buffer starting at *p* of length *len.*

~filebuf

The filebuf destructor is defined as follows:

virtual ~filebuf()

Class filebuf Member Functions

The following public member functions are defined in class filebuf in include file *fstream.h.*

attach

filebuf* attach(int *filedescriptor*);

Attaches the filebuf object to an open file descriptor.

close

filebuf* close();

Flush and close the attached file descriptor. Clears error state unless error on close.

fd

int fd();

Returns the attached file descriptor; returns end-of-file if file is closed.

is_open

int is_open();

Returns nonzero if filebuf is connected to a file descriptor; otherwise, returns 0.

open

filebuf * open(char* *filename*, **int** *omode*, **int** *protection*=**openprot);**

Open filename in appropriate mode and protection and connect to filebuf.

Class strstreambuf

The class strstreambuf is derived from streambuf and supports storing and fetching from incore arrays of characters. The incore array can be a static array or a dynamically allocated and resized array.

Class strstreambuf—Constructors and Destructor

strstreambuf

The following constructors are defined in class strstreambuf:

strstreambuf();

Construct an empty strstreambuf in dynamic mode.

strstreambuf(char* *p*, int *n*, char* *pstart*);

Construct an empty strstreambuf in static mode starting at *p* according to the following rules:

if *n* is positive: *n* is the length of the buffer.

if *n* is zero: *p* is taken to be a null terminated string.

if *n* is negative: no end is specified.

The get pointer is initialized to *p*, the put pointer is initialized to *pstart*.

strstreambuf(int *len*);

Construct an empty strstreambuf in dynamic mode with initial buffer allocation of at least length *len*.

strstreambuf(void* (alloc_func*)(long), void (**free_func*)(void*));**

Construct an empty strstreambuf in dynamic mode. If *alloc_func* is non-null use it to allocate the buffer instead of **new()**. If *free_func* is non-null use it to free the buffer instead of **delete()**.

strstreambuf(unsigned char* *b*, int *size*, unsigned char* *pstart* = 0) ;

Construct an empty strstreambuf in static mode starting at *p* according to the following rules:

If *n* is positive: n is the length of the buffer.

If *n* is zero: *p* is taken to be a null terminated string.

If *n* is negative: no end is specified.

The get pointer is initialized to *p*, the put pointer is initialized to *pstart*.

strstreambuf

The strstreambuf destructor.

virtual ~strstreambuf();

Class strstreambuf - Public Functions

freeze

void freeze(int *n* = 1);

If *n* is 0, permits automatic deletion of the current dynamic array.

If *n* is nonzero, automatic deletion is not permitted.

str

char * str();

Returns a pointer to the current array.

Class stdiobuf

The stdiobuf class is derived from streambuf and supports I/O through **FILE** structs. It is intended for mixing C and C++ code and should be avoided if possible. These functions are available by including *stdiostream.h*.

Note

This class is obsolete and should not be used.

Manipulators

The include files *iomanip.h* and *iostream.h* contain functions that take an ios&, istream& or ostream& and return their argument. Such functions are called *manipulators*. These functions also change the format state information.

dec

outstream << dec
instream >> dec

Change formatting conversion base to decimal.

hex

outstream << hex
instream >> hex

Change formatting conversion base to hexadecimal.

oct

outstream << oct
instream >> oct

Change formatting conversion base to octal.

ws

instream >> ws

Skip over next whitespace in input stream.

setw

outstream << setw(*n*)
instream >> setw(*n*)

Set field width to *n*.

setfil

outstream << setfill(*n*)
instream >> setfill(*n*)

Set fill character to *n*.

setprecision

outstream << setprecision(*n*)
instream >> setprecision(*n*)

Set precision to *n*.

setiosflags

outstream << setiosflags(long *n*)
instream >> setiosflags(long *n*)

Set the format flags indicated in *n*.

resetiosflags

outstream << resetiosflags(long *n*)
instream >> resetiosflags(long *n*)

Clear the format flags indicated in *n*.

endl

outstream << endl

End line. That is, insert \n and flush.

ends

outstream << ends

End string. Insert \0.

flush

outstream << flush

Flush stream.

The C++ Complex Math Library

The header file *complex.h* and the C++ library *libcmplx.a* provide definitions and support for complex arithmetic in C++.

The complex numeric data type is implemented as a new data type, complex, with operators and functions. The complex class is defined to hold a single complex number consisting of a real part and an imaginary part. The real and imaginary parts are represented in the class complex as a pair of doubles.

Complex Constructor and Destructor

complex

The complex class provides two constructors.

void complex();

The first takes no arguments and declares a complex variable that is initialized so that both its real and imaginary parts are initialized to 0.0.

complex (double *r*, double *i* = 0.0);

The second constructor takes one or two arguments; the first argument initializes the real component of the complex object, and the second argument initializes the imaginary part. When only one argument is given, a default argument of 0.0 is supplied for the second argument. The supplied value initializes the real part and the imaginary part is initialized to zero.

Below are several examples showing use of the complex constructors.

```
complex x;
```

This creates a complex object named `x` which is initialized to (0.0,0.0), with its real and imaginary parts both initialized to zero.

```
complex y(9.9,1.1);
```

This creates a complex object named y which is initialized to (9.9,1.1). This represents 9.9 + 1.1*i*.

```
complex z(8.4);
```

This uses the second constructor to create a complex object named `z` which is initialized to (8.4,0.0).

~complex

The complex class does not provide a special destructor function.

Complex Arithmetic Operators

The following operators are overloaded to provide complex operations using the usual precedence rules.

+	addition
-	subtraction
-	unary minus
*	multiplication
/	division
=	assignment
==	equals
!=	not equal

The following operators are overloaded, but may not be used to return a value in an expression.

+=	Compound assignment (addition)
-+	Compound assignment (subtraction)
*=	Compound assignment (multiplication)
/=	Compound assignment (division)

For example, the following are valid complex assignments:

```
complex comp1, comp2;
comp1 += 1;
comp2 -= comp1;
```

Complex Public Functions

This section shows the public complex math functions provided with **iCC**.

abs

double abs(complex);

Returns the magnitude of the complex argument.

arg

double arg(complex);

Returns the phase angle of the complex argument.

conj

complex conj(complex);

Returns the complex conjugate of the complex argument.

cos

complex cos(complex);

Returns the cosine of the complex argument.

cosh

complex cosh(complex);

Returns the hyperbolic cosine of the complex argument.

exp

complex exp(complex);

Returns e**x where e is 2.718281828, and x is the complex argument.

imag

double imag(const complex&);

Returns the imaginary part of the complex argument.

log

complex log(complex);

Returns the natural logarithm of the complex argument. If the argument is 0, this causes an error and the value returned in the real part is a very large number.

norm

double norm(complex);

Returns the square of the magnitude of the complex argument.

polar

complex polar(double, double = 0);

Returns a complex number with the given magnitude (first argument) and phase angle (second argument).

pow

complex pow(double, complex);

Returns a ** b where a is the first argument (a double) and b is the second argument (a complex).

complex pow(complex, int);

Returns a ** b where a is the first argument (a complex) and b is the second argument (an int).

complex pow(complex, double);

Returns a ** b where a is the first argument (a complex) and b is the second argument (a double).

complex pow(complex, complex);

Returns a ** b where a is the first argument (a complex) and b is the second argument (a complex).

real

double real(const complex&);

Returns the real part of the complex argument.

sin

complex sin(complex);

Returns the sine of the complex argument.

sinh

complex sinh(complex);

Returns the hyperbolic sine of the complex argument.

sqrt

complex sqrt(complex);

Returns the square root of the complex argument.

Input and Output Using Complex Values

For the complex class, the operators << and >> are overloaded to provide input and output of complex numbers (see the Iostream library description for more details). Input is expected in the form (x,y) where x and y are doubles. For example:

```
complex cval1;
cin >> cval1;
```

with the input values:

```
(100.0, 40.0)
```

gives cval1 the value 100.0 + 40i.

For output, the real and imaginary parts of the number are enclosed in parentheses and the "i", for imaginary, is not included in the imaginary number.

```
complex cval2(12.0,3.0);
cout << cval2;
```

produces the following output:

```
(12.0,3.0)
```

Error Handling

The complex class includes a mechanism for handling errors, using the errno integer flag and the integer function **complex_error(int, double)**.

The error handler function sets the value of the error flag errno. The library defines several values for error conditions for the functions **cosh**, **exp**, **log**, and **sinh**.

The **cosh** error values are:

`C_COSH_RE` The real part was too large.

`C_COSH_IM` The imaginary part was too large.

The **exp** error values are:

`C_EXP_RE_POS`
The imaginary part was too small.

`C_EXP_RE_NEG`
The real part was too small.

The **log** error values are:

`C_LOG_0` The real and imaginary parts were 0.

The **sinh** error values are:

`C_SINH_RE` The real part was too large.

`C_SINH_IM` The imaginary part was too large.

Template Instantiation 7

A template defines a family of types or functions. For example, the following code fragment shows a template declaration of a class **vect**. This template declaration can be used to declare vector objects. By supplying different types for the parameter T, different template class definitions are instantiated or generated.

```
// define a template
template<class T> class vect{
  private:
    T * v;
    typedef int vect_index_t;
    vect_index_t size;
  public:
    vect(vect_index_t x) { size = x; v = new int[x];};
    T& operator[](vect_index_t);
    T& element(vect_index_t i);
};

template<class T> T& vect<T>::element(vect_index_t i)
{
  return v[i];
}
```

The following program fragment shows the template class **vect** being used.

```
// use the template
vect<int> x(80);
vect<double> d(20);

void foo(void)
{
  int j = x.element(5);
  double f = d.element(6);
}
```

The previous program requires two instantiations of template class **vect**: one where T is **int** and one where T is **double**. It would seem the compiler could just generate these instantiations, but unfortunately things are not that simple.

If the template declaration of class **vect** was in an include file, and another module included it and used it, other instantiations of this template might be needed. Another module might also require an instantiation where T is **int**. In this case, we would like only one instantiation of template class **vect** where T is **int**.

C++ also allows specialization of a template entity. This is a type specific version to be used in place of the version that would have been generated from the template. In the above example, someone could write a specialization for type **int**:

```
int& vect<int>::element(vect_index_t i)
{
// check bounds for int vectors
   if (i >= size){
         extern void error(char *);
         error("vect index out of bounds\n");
   }
   return v[i];
}
```

In this case, we would like to use the specialized member function element.

C++ also dictates that unreferenced template functions should not be compiled.

For these reasons, the compiler cannot know what instantiations are required or in which modules to put them until the whole program is linked. The programmer should have an idea where these templates should be expanded. First we discuss two methods the programmer can use to tell the compiler where to put template instantiations. Then we discuss an automatic instantiation scheme.

Command Line Control

Normally, when a file is compiled, no template entities are instantiated except those assigned to the file by automatic instantiation (see the section "Automatic Template Instantiation" on page 7-3).

The overall instantiation mode can, however, be changed by the following command-line options:

-tnone Do not automatically create instantiations of any template entities. This is the default. It is also the usually appropriate mode when automatic instantiation is done.

-tused Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions.

-tall 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 will be instantiated whether or not they were used. Nonmember template functions will be instantiated even if the only reference was a declaration.

-tlocal Similar to **-tused** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables.) However, one may end up with many copies of the instantiated functions. **–tlocal** cannot be used in conjunction with automatic template instantiation.

Automatic Template Instantiation

The goal of an automatic instantiation mode is to provide painless instantiation. The programmer should be able to compile source files to object code, then link them and run the resulting program, and never have to worry about how the necessary instantiations get done. In practice, this is hard for a compiler to do.

Our approach requires that for each instantiation required, there is some (normal, top-level, explicitly-compiled) source file that contains both the definition of the template entity and of any types required for the particular instantiation.

This is not always the case. Suppose that file A contains a definition of class **X** and a reference to **Stack<X>::push**, and that file B contains the definition for the member function **push**. There would be no file containing both the definition of **push** and the definition of **X**.]

This requirement can be met in various ways:

1. Each *.h* file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.

2. Implicit inclusion: when the compiler sees a template declaration in a *.h* file and discovers a need to instantiate that entity, it is given permission to go off looking for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method allows most programs written using the cfront convention to be compiled (see the section "Implicit Inclusion" on page 7-5).

3. The ad hoc approach: the programmer makes sure that the files that define template entities also have the definitions of all the available types, and adds code or pragmas in those files to request instantiation of the entities there.

The automatic instantiation method works as follows:

1. The first time the source files of a program are compiled, no template entities are instantiated. However, the generated object files contain information about things that could have been instantiated in each compilation.

2. When the object files are linked together, a special pre-linker program called **iprelnk** is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.

3. If **iprelnk** finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it assigns the instantiation to it. The set of instantiations assigned to a given file, say *abc.C*, is recorded in an associated *.ii* file, for example, *abc.ii*.

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

5. When the compiler compiles a file, it reads the *.ii* file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).

6. **iprelnk** repeats steps 3-5 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. From then on, whenever source files are recompiled, the compiler will consult the *.ii* files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the **iprelnk** step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done. That's true even if the entire program is recompiled.

If the programmer provides a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the **iprelnk** step. Since that definition satisfies whatever references there might be to that entity, the **iprelnk** program will see no need to request an instantiation of the entity. If the programmer adds a specialization to a program that has previously been compiled, the **iprelnk** program will notice that too and remove the assignment of the instantiation from the proper *.ii* file.

The *.ii* files should not, in general, require any manual intervention. One exception: if a definition is changed in such a way that some instantiation no longer compiles (it gets errors), and at the same time a specialization is added in another file, and the first file is being recompiled before the specialization file and is getting errors, the *.ii* file for the file getting the errors must be deleted manually to allow **iprelnk** to regenerate it.

If **iprelnk** changes an instantiation assignment, it will issue a message like the following:

```
C++ iprelnk: f__10A__pt__2_iFv assigned to file test.o
C++ iprelnk: executing: /usr/pgi/i860/bin/pgCC -c test.c
```

The name in the message is the mangled name of the entity.

The automatic instantiation scheme can coexist with partial explicit control of instantiation by the programmer through the use of pragmas or command-line specification of the instantiation mode.

Implicit Inclusion

Implicit inclusion is provided in order to facilitate existing cfront programs.

When implicit inclusion is enabled with the **-Wc,-B** command line switch, the front end is given 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 will process it as if it were included at the end of the main source file.

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

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done.

C++ Name Mangling 8

Name mangling transforms the names of entities so that the names include information on aspects of the entity's type and fully qualified name. This is necessary since the intermediate language into which a program is translated contains fewer and simpler name spaces than there are in the C++ language. Specifically:

- Overloaded function names are not allowed in the intermediate language.
- Classes have their own scopes in C++, but not in the generated intermediate language. For example, an entity x from inside a class must not conflict with an entity x from the file scope.
- External names in the object code form a completely flat name space. The names of entities with external linkage must be projected onto that name space so that they do not conflict with one another. A function f from a class A, for example, must not have the same external name as a function f from class B.
- Some names are not names in the conventional sense of the word, they're not strings of alphanumeric characters, for example operator=.

There are two problems here:

1. Generating external names that will not clash.
2. Generating alphanumeric names for entities with strange names in C++.

Name mangling solves these problems by generating external names that will not clash, and alphanumeric names for entities with strange names in C++. It also solves the problem of generating hidden names for some behind-the-scenes language support in such a way that they will match up across separate compilations.

You will see mangled names if you view files that are translated by **iCC**, and you do not use tools that demangle the C++ names. Intermediate files that use mangled names include the assembly and object files created by **iCC.**

The name mangling algorithm for **iCC** is the same as that for **cfront** and also matches the description in Section 7.2, "Function Name Encoding" of the ARM (except for some minor details). Refer to the ARM for a complete description of name mangling.

Types of Mangling

The following entity names are mangled:

- **Function names** including non-member function names are mangled, to deal with overloading. Names of functions with extern "C" linkage are not mangled.

  Mangled function names have the function name followed by __ followed by F followed by the mangled description of the types of the parameters of the function. If the function is a member function, the mangled form of the class name precedes the F. If the member function is static, an S also precedes the F.

```
int f(float);                 // f__Ff
class A
  int f(float);               // f__1AFf
  static int g(float);        // g__1ASFf
;
```

- **Special and operator function names**, like constructors and **operator=()**. The encoding is similar to that for normal functions, but a coded name is used instead of the routine name:

```
class A
  int operator+(float);       // __pl__1AFf
  A(float);                   // __ct__1AFf
;
int operator+(A, float);      // __pl__F1Af
```

- **Static data member names.** The mangled form is the member name followed by __ followed by the mangled form of the class name:

```
class A
  static int i;               // i__1A
;
```

- **Names of variables generated for virtual function tables.** These have names like:

```
vtblmangled-class-name or
vtblmangled-base-class-namemangled-class-name.
```

- **Names of variables generated to contain runtime type information**. These have names like:

```
Ttype-encoding and TIDtype-encoding.
```

Mangling Summary

This section lists some of the C++ entities that are mangled and provides some details on the mangling algorithm. For more details, refer to *The Annotated C++ Reference Manual.*

Type Name Mangling

Using **iCC**, each type has a corresponding mangled encoding. For example, a class type is represented as the class name preceded by the number of characters in the class name, as in **5abcde** for **abcde**. Simple types are encoded as lower-case letters, as in **i** for **int** or **f** for **float**. Type modifiers and declarators are encoded as upper-case letters preceding the types they modify, as in **U** for **unsigned** or **P** for **pointer**.

Nested Class Name Mangling

Nested class types are encoded as a Q followed by a digit indicating the depth of nesting, followed by a _, followed by the mangled-form names of the class types in the fully-qualified name of the class, from outermost to innermost:

```
class A
  class B   // Q2_1A1B
  ;
;
```

Local Class Name Mangling

The name of the nested class itself is mangled to the form described above with a prefix __, which serves to make the class name distinct from all user names.

Local class names are encoded as L followed by a number (which has no special meaning; it's just an identifying number assigned to the class) followed by __ followed by the mangled name of the class (this is not in the ARM, and **cfront** encodes local class names slightly differently):

```
void f()
  class A   // L1__1A}
  ;
;
```

This form is used when encoding the local class name as a type. It's not necessary to mangle the name of the local class itself unless it's also a nested class.

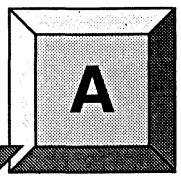
Template Class Name Mangling

Template classes have mangled names that encode the arguments of the template:

```
template<class T1, class T2> class abc ;
abc<int, int> x;
abc__pt__3_ii
```

This describes two template arguments of type **int** with the total length of template argument list string, including the underscore, and a fixed string, indicates parameterized type as well, the name of the class template.

Compiler Error Messages



This appendix lists the error messages generated by the Paragon System C++ compiler, indicating each message's severity and, where appropriate, the error's probable cause and correction. In the error messages, the dollar sign ($) represents information that is specific to each occurrence of the message. Note that messages issued by the C++ front-end are not currently included in this list.

Each error message is numbered and preceded by one of the following letters, indicating its severity:

I	Informative.
W	Warning.
S	Severe error.
F	Fatal error.
V	Variable.

`V000  Internal compiler error. $ $`

This message indicates an error in the compiler. The severity may vary; if it is informative or warning, the compiler probably generated correct object code, but there is no way to be sure. Regardless of the severity, please report any internal error to Intel Supercomputer Systems Division Customer Support.

`F001  Source input file name not specified`

On the command line, source file name should be specified either before all the switches, or after them.

F002 Unable to open source input file: $

Source file name misspelled, file not in current working directory, or file is read protected. Also can be issued if include file is read protected.

F003 Unable to open listing file

Probably, user does not have write permission for the current working directory.

F004 Unable to open object file

Probably, user does not have write permission for the current working directory.

F005 Unable to open temporary file

Compiler uses directory */usr/tmp* or */tmp* in which to create temporary files. If neither of these directories is available on the node on which the compiler is being used, this error will occur.

F006 Empty translation unit

Source input file does not contain any declarations or function definitions.

F007 Source file too large to compile at this optimization level

Symbol table overflowed, or compiler working storage space exhausted. If this error occurred at optimization level 2, reducing the optimization level to 1 may work around the problem, otherwise splitting the source file in two should be considered. There is no hard limit on how large a file the compiler can handle, but as a very rough estimate, if the file is less than 2000 lines long (not counting comments), and this error occurs, it may represent a compiler problem.

I009 <reserved message number>

I010 <reserved message number>

S011 Unrecognized command line switch: $

Refer to the Chapter 2 of this manual for a list of the allowed switches.

S012 Value required for command line switch: $

Certain switches require a value which immediately follows, such as **-O 2**.

S013 Unrecognized value specified for command line switch: $

S014 Ambiguous command line switch: $

Too short an abbreviation was used for one of the switches.

I015 <reserved message number>

I017 <reserved message number>

I018 <reserved message number>

S073 Too many initializers for $

The initializer for an array or structure contains too many constants.

W085 Truncation performed for field initialization

An integer constant used to initialize a structure field is too large for the field.

S086 Division by zero

A division by zero was encountered while constant folding a constant expression.

W106 Shift count out of range

The bit count for a shift operation must be in the range 0 to 31. Note that a shift count of 32 will not produce a result of zero on some machines.

W116 Constant value out of range for signed short or char

Note that a constant such as **0xFFFF** (**0xff**), interpreted as a positive number, is 1 bit too large for the signed **short** (**char**) data type. Either the type **unsigned short** (**unsigned char**) should be used in place of signed **short** (**char**), or the equivalent negative number should be used in place of the positive constant.

W129 Floating point overflow. Check constants and constant expressions

W130 Floating point underflow. Check constants and constant expressions

W131 Integer overflow. Check floating point expressions cast to integer

S132 Floating pt. invalid oprnd. Check constants and constant expressions

S133 Divide by 0.0. Check constants and constant expressions

W199 Unaligned memory reference

A memory reference occurred whose address does not meet its data alignment requirement.

W271 Can't inline $ - wrong number of arguments

I272 Argument of inlined function not used

S273 Inline library not specified on command line (-inlib switch)

F274 Unable to access file $/TOC

S275 Unable to open file $ for inlining

Manual Pages B

This appendix contains manual pages for compiler-related commands and system calls.

- See the *OSF/1 Command Reference* and *OSF/1 Programmer's Reference* for manual pages for the standard commands and system calls of OSF/1.
- See the *Paragon*™ *System Commands Reference Manual* and the *Paragon*™ *System C Calls Reference Manual* for manual pages for parallel commands and system calls unique to the operating system for the Paragon™ system.

The manual pages in this appendix are also available on-line, using the **man** command.

Table B-1. lists the commands described in this appendix.

Table B-1. Commands Discussed in This Appendix

Manual Page	Commands	Description
ar860	**ar860** (cross) **ar** (native)	Manages object code libraries.
as860	**as860** (cross) **as** (native)	Assembles i860™ source code.
cpp860	**cpp860** (cross) **cpp** (native)	Preprocesses C programs.
dump860	**dump860** (cross and native)	Dumps object files.
iCC	**iCC** (cross) **CC** (native)	Invoked the C++ compiler.
ld860	**ld860** (cross) **ld** (native)	Links object files.
mac860	**mac860** (cross) **mac** (native)	Preprocesses assembly-language programs.
nm860	**nm860** (cross) **nm** (native)	Displays symbol table (name list) information.
size860	**size860** (cross) **size** (native)	Displays section sizes of object files.
strip860	**strip860** (cross) **strip** (native)	Strips symbol information from object files.

Except for their names, the cross-development and native versions of each command work the same (with minor exceptions). These commands are available by their cross-development names on the Paragon system and on supported workstations; they are available by their native names on the Paragon system only.

AR860 AR860

ar860, ar: Creates and maintains archives for the Paragon™ system.

Cross-Development Syntax

ar860 [**-V**] *key* [*options*] *libname* [*filename ...*]

Native Syntax

ar [**-V**] *key* [*options*] *libname* [*filename ...*]

Arguments

libname The name of the archive.

filename The name of the target file.

You must specify one, and only one, *key* from the following list:

d Delete *filename* from the archive.

e Display the symbol tables of COFF objects in the archive.

p Display the archive version of *filename* (may result in binary data being sent to standard output).

q Quickly add the file *filename* to the archive *libname* by appending the file(s) to the end of the archive without checking to see if they duplicate existing files in the archive. If *libname* does not exist, then create it (unless the **c** option is specified). If *filename* does not appear in the archive, then add it.

r Replace the file *filename* in the archive *libname*. If *libname* does not exist, then create it. If *filename* does not appear in the archive, then add it.

t Display the archive table of contents.

x Extract *filename* from the archive. If no file is named, extract all files.

The *key* argument may be preceded by a dash. For example, **ar860 -t file.a** and **ar860 t file.a** are equivalent.

AR860 *(cont.)* AR860 *(cont.)*

You may specify the following *options* in any order:

c	Suppress the creation message. This option is used with the **-r** key.
l	Use the current working directory for temporary files.
u	Replace the archive version only if filename is newer. This option is used only with the **-r** key.
v	Verbose mode. For **-r**, display the names of the archive members as they are replaced (or added). For **-d**, display the names of the archive members as they are deleted. For **-t**, display the file mode, the *uid*, the *gid*, the size, and the timestamp of the specified files. For **-x**, display the names of the files as they are extracted.

No space may appear between the *key* and any *options*.

You must specify the following argument, if used, before the *key*:

-V	Display the tool banner (tool name, version, etc.).

No space may appear between **-V** and the following *key*, and the *key* may not be preceded by a dash. The dash preceding the **V** is optional. For example, **ar860 -Vt file.a** and **ar860 Vt file.a** are equivalent.

Description

Use **ar860** to manage archives for the Paragon system.

See Also

as860, dump860, icc, if77, ld860, nm860, size860, strip860

AS860 AS860

as860, as: Assembles i860 code for the Paragon™ system.

Cross-Development Syntax

as860 [*switches*] [*filename*]

Native Syntax

as [*switches*] [*filename*]

Arguments

filename The name of the i860 assembly language file. If no file is specified, **as860** reads from standard input.

You may specify the following *switches* in any order:

-a Do not automatically import symbols that are referenced but otherwise undefined. Issues an error message for each occurrence.

-l[*listfile*] Write source listing in the file *listfile*, a file in the current working directory. If you omit *listfile*, the listing goes to standard output.

-L Preserve text symbols starting with ".L" in the debug section.

-o *objfile* Put the output object file in *objfile*. If you omit this switch, the default object file name is produced by stripping any directory prefixes from *filename*, stripping any of the suffixes ".n10", ".s", ".mac", or ".860", and appending ".o". An existing file with the same name is silently overwritten.

-R Suppress all **.data** directives. Code and data are both assembled into the **.text** section.

-V Display the tool banner (tool name, version, etc.).

-x Enable additional checks of the source file to find illegal sequences of instructions.

AS860 *(cont.)*

AS860 *(cont.)*

Description

Use **as860** to assemble the named file.

You can ensure that the proper switches are passed to **as860** by accessing **as860** using the compiler drivers (**icc** or **if77**).

Not all illegal sequences are detected when the **-x** switch is used.

See Also

ar860, dump860, icc, if77, ld860, nm860, size860, strip860

DUMP860 DUMP860

Dumps parts of a Paragon™ system object file.

Syntax

dump860 [*switches*] *filename*

Arguments

filename The name of the object file.

You may specify the following *switches* in any order:

-a Display archive headers.

-c Dump the string table.

-d *number* Dump section headers starting at section *number*. Only effective if the **-h** switch is also specified. Sections are numbered starting at 1. If the **+d** switch is not specified, then only the single section header is dumped.

+d *number* Dump section headers ending at section *number*. Only effective if the **-h** switch is used.

-f Display file headers.

-g Display the archive symbol table.

-h Dump section headers.

-l Dump line numbers.

-n *name* Dump only sections named *name*. Only effective if the **-h** switch is used.

-o Dump (in formatted hexadecimal) optional headers.

-p Do not display headers.

-r Dump relocation data.

-s Dump section data.

DUMP860 *(cont.)*

DUMP860 *(cont.)*

-t [*number*]	Dump symbol table, starting at symbol index *number*. If the **+t** switch is not used, then only the single symbol is displayed.
+t *number*	Dump symbol table, through symbol index *number*. If **-t** was not specified, the start index is zero.
-u	Underline mode. Only works on devices supporting backspace.
-v	Verbose mode. Display some headers and information in an easier-to-comprehend form.
-V	Display the tool banner (tool name, version, etc.).
-z *name,number*	Dump line numbers for function *name*, starting at line *number*.
+z *number*	Dump line numbers for function *name* (specified by **-z**), ending at line *number*.

Description

Use **dump860** to dump (in formatted hexadecimal) parts of the named object file.

See Also

ar860, as860, icc, if77, ld860, nm860, size860, strip860

ICC ICC

iCC, CC: Invokes the Paragon™ system C++ compiler

Cross-Development Syntax

iCC [*options*] *filename ...*

Native Syntax

CC [*options*] *filename ...*

Arguments

options	Any of the command-line options listed in the following section.
filename	The name of the source file.

Options

-A	Accepts the proposed version of ANSI C++.
-b	Compiles with cfront compatibility, version 2.1.
-b3	Compiles with cfront compatibility, version 3.0.
-c	Skips the link step. Compile and assembles only (to file.o for each file.c)
-C	Preserves comments in preprocessed source files. Also enables **-E**.
-dryrun	Shows all commands created by the driver but does not execute any commands.
-D*name*[=*def*]	Defines *name* to be *def* in the preprocessor. If *def* is missing, it is assumed to be empty. If the = sign is also missing, then *name* is defined to be the string `1`.
-e	Sets the error limit.
-E	Preprocesses very input file, regardless of suffix, and sends the result to *stdout*. No compilation, assembly, or linking is performed.
-flags	Displays a list of all valid driver options.
-help	Displays a list of all valid driver options.

ICC *(cont.)*

ICC *(cont.)*

-I*directory* Adds *directory* to the compiler's search path for include files. If you use more than one **-I** switch, the specified directories are searched in the order they were specified (left to right). For include files surrounded by angle brackets (<...>), each *directory* is searched followed by the default location. For include files surrounded by double quotes ("..."), the directory containing the file containing the **#include** directive is searched, followed by the **-I** directories, followed by the default location.

-K*option* Requests special mathematical semantics. The *option* values are:

ieee (default) If used while linking, links in a math library that conforms with the IEEE 754 standard.

If used while compiling, tells the compiler to perform **float** and **double** divides in conformance with the IEEE 754 standard.

ieee=enable If used while linking, has the same effects as **-Kieee**, and also enables floating-point traps and underflow traps. If used while compiling, has the same effects as **-Kieee**.

ieee=strict If used while linking, has the same effects as **-Kieee=enable**, and also enables inexact traps. If used while compiling, has the same effects as **-Kieee**.

noieee If used while linking, produces a program that flushes denormals to 0 on creation, which reduces underflow traps. If used together with **-lm**, also links in a version of *libm.a* that is not as accurate as the standard library, but offers greater performance. This library offers little or no support for exceptional data types such as **INF** and **NaN**, and will trap on such values when encountered.

If used while compiling, tells the compiler to perform **float** and **double** divides using an inline divide algorithm that offers greater performance than the standard algorithm. This algorithm produces results that differ from the results specified by the IEEE standard by no more than three units in the last place.

ICC *(cont.)*

ICC *(cont.)*

trap=fp If used while linking, disables kernel handling of floating-point traps. Has no effect if used while compiling.

trap=align If used while linking, disables kernel handling of alignment traps. Has no effect if used while compiling.

-l*library* Load the library **lib***library***.a**. The library is loaded from the first library directory in the library search path (see the **-L** switch) in which a file of that name is encountered. (Passed to the linker.)

-L*directory* Adds *directory* to beginning of the library search path. Also see the **nostdlib** and **nostartup** options of the **-M** switch. (Passed to the linker; see the **ld860** manual page for more information on the library search path.)

-m Produces a link map. (Passed to the linker.)

-M*option* Requests specific actions from the compiler. The *option* values are as follows (an unrecognized **-M** *option* is passed directly to the compiler):

anno Produce annotated assembly files, where source code is intermixed with assembly language. **-Mkeepasm** or **-S** should be used as well.

[no]bounds [Don't] enable array bounds checking (default **-Mnobounds**). With **-Mbounds** enabled, bounds checking is not applied to subscripted pointers or to externally-declared arrays whose dimensions are zero (**extern arr[]**). Bounds checking is not applied to an argument even if it is declared as an array. If an array bounds checking violation occurs when a program is executed, an error message describing where the error occurred is printed and the program terminates. The text of the error message includes the name of the array, where the error occurred (the source file and line number in the source), and the value, upper bound, and dimension of the out-of-bounds subscript. The name of the array is not included if the subscripting is applied to a pointer.

[no]dalign [Don't] align **double**s in structures on double-precision boundaries (default **-Mdalign**). **-Mnodalign** may lead to data alignment exceptions.

ICC *(cont.)*

ICC *(cont.)*

[no]debug [Don't] generate symbolic debug information (default **-Mnodebug**). If **-Mdebug** is specified with an optimization level greater than zero, line numbers will not be generated for all program statements. **-Mdebug** increases the object file size.

[no]depchk [Don't] check for potential data dependencies exist (default **-Mdepchk**). This is especially useful in disambiguating unknown data dependencies between pointers that cannot be resolved at compile time. For example, if two floating-point array pointers are passed to a function and the pointers never overlap and thus never conflict, then this switch may result in better code. The granularity of this switch is rather coarse, and hence the user must use precaution to ensure that other *necessary* data dependencies are not overridden. Do not use this switch if such data dependencies do exist. **-Mnodepchk** may result in incorrect code; the **-Msafeptr** switch provides a less dangerous way to accomplish the same thing.

[no]frame [Don't] include the frame pointer (default **-Mnoframe**). Using **-Mnoframe** can improve execution time and decrease code, but makes it impossible to get a call stack traceback when using a debugger.

[no]func32 [Don't] align functions on 32-byte boundaries (default **-Mfunc32**). **-Mfunc32** may improve cache performance for programs with many small functions.

info=[*option*[,*option*...]]
Produce useful information on the standard error output. The options are:

time or **stat** - Output compilation statistics.

loop - Output information about loops. This includes information about vectorization and software pipelining.

inline - Output information about functions extracted and inlined.

ICC *(cont.)*

ICC *(cont.)*

inline=[*option*[,*option*...]]
Pass options to the function inliner. The *options* are:

levels:*number* - Perform *number* levels of inlining (default 0).

See Chapter 3 for more information on using the compiler's function inliner.

keepasm Keep the assembly file for each source file, but continue to assemble and link the program.

nolist Don't create a listing file (this is the default).

[no]longbranch [Don't] allow compiler to generate **bte** and **btne** instructions (default **-Mlongbranch**). **-Mnolongbranch** should be used only if an assembly error occurs.

nostartup Don't link the usual start-up routine (*crt0.o*), which contains the entry point for the program.

nostddef Don't predefine any system-specific macros to the preprocessor when compiling a C program. (Does not affect ANSI-standard preprocessor macros.) The system-specific predefined macros are **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, and **__NODE** (**__NODE** is only defined when compiling with **-nx**). See also **-U**.

nostdinc Remove the default include directory (*/usr/include* for **CC**, *$(PARAGON_XDEV)/paragon/include* for **iCC**) from the include files search path.

nostdlib Don't link the standard libraries (*libpm.o*, *guard.o*, *libc.a*, *iclib.a*, and *libmach3.a*) when linking a program.

[no]perfmon [Don't] link the performance monitoring module (*libpm.o*) (default **-Mperfmon**). See the *Paragon™ System Application Tools User's Guide* for information on performance monitoring.

ICC *(cont.)*

ICC *(cont.)*

[no]quad [Don't] force top-level objects (such as local arrays) of size greater than or equal to 16 bytes to be quad-aligned (default **-Mquad**). Note that **-Mquad** does not affect items within a top-level object; such items are quad-aligned only if appropriate padding is inserted.

[no]reentrant [Don't] generate reentrant code (default **-Mreentrant**). **-Mreentrant** disables certain optimizations that can improve performance but may result in code that is not reentrant. Even with **-Mreentrant**, the code may still not be reentrant if it is improperly written (for example, if it declares static variables).

safeptr=[*option*[,*option*...]]
Override data dependence between C++ pointers and arrays. This is a potentially very dangerous option since the potential exists for code to be generated that will result in unexpected or incorrect results as is defined by the ANSI C++ working draft. However, when used properly, this option has the potential to greatly enhance the performance of the resulting code, especially floating-point oriented loops. Combinations of the *options* can be used.

dummy or **arg** - C++ dummy arguments (pointers and arrays) are treated with the same copyin/copyout semantics as Fortran dummy arguments.

auto - C++ local or **auto** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.

static - C++ **static** variables (pointers and arrays) are assumed to not overlap or conflict with each other and to be independent.

global - C++ global or **extern** variables (pointers and arrays) are assumed not to overlap or conflict with each other and are independent.

ICC *(cont.)*

ICC *(cont.)*

[no]signextend [Don't] sign extend when a narrowing conversion overflows (default **-Msignextend**). For example, if **-Msignextend** is in effect and an integer containing the value 65535 is converted to a **short**, the value of the **short** will be -1. This option is provided for compatibility with other compilers, even though ANSI C specifies that the result of such conversions are undefined. **-Msignextend** will decrease performance on such conversions.

[no]streamall [Don't] stream all vectors to and from cache in a vector loop (default **-Mstreamall**). When **-Mnostreamall** is in effect, the compiler chooses one vector to come directly from or go directly to main memory, without being streamed into or out of cache.

[no]stride0 [Don't] inhibit certain optimizations and allow for stride 0 array references. **-Mstride0** may degrade performance, and should only be used if zero stride induction variables are possible. (default **-Mnostride0**).

vect[=*option*[,*option*...]]

Perform vectorization (also enables **-Mvintr**). If no *option*s are specified, then all vector optimizations are enabled. The available *option*s are:

cachesize:*number* - This sets the size of the portion of the cache used by the vectorizer to *number* bytes. *Number* must be a multiple of 16, and less than the cache size of the microprocessor (16384 for the i860 XP, 8192 for the i860 XR). In most cases the best results occur when *number* is set to 4096, which is the default (for both microprocessors).

noassoc - When scalar reductions are present (for example, dot product), and loop unrolling is turned on, the compiler may change the order of operations so that it can generate better code. This transformation can change the result of the computation due to round-off error. The use of **noassoc** prevents this transformation.

recog - Recognize certain loops as simple vector loops and call a special routine.

ICC *(cont.)*

ICC *(cont.)*

smallvect[*:number*] - This option allows the vectorizer to assume that the maximum vector length is no greater than *number*. *Number* must be a multiple of 10. If *number* is not specified, the value 100 is used. This option allows the vectorizer to avoid stripmining in cases where it cannot determine the maximum vector length. In doubly-nested, non-perfectly nested loops this option can allow invariant vector motion that would not otherwise have been possible. Incorrect code will result if this option is used, and a vector takes on a length greater than specified.

streamlim:*n* - This sets a limit for application of the vectorizer data streaming optimization. If data streaming requires cache vectors of length less than *n*, the optimization is not performed. Other vectorizer optimizations are still performed. The data streaming optimization has a high overhead compared to other loop optimizations, and can be counter-productive when used for short vectors. The *n* specifier is not optional. The default limit is 32 elements if **streamlim** is not used.

transform - Perform high-level transformations such as loop splitting and loop interchanging. This is normally not useful without **-Mvect=recog**.

-Mvect with no options means **-Mvect=recog,transform,cachesize:4096**.

[no]xp [Don't] use i860 XP microprocessor features (default **-Mxp**).

-nostdinc Equivalent to **-Mnostdinc**.

-nx Creates an executable application for multiple nodes.

- If used while compiling, it defines the preprocessor symbol **__NODE**. The program being compiled can use preprocessor statements such as **#ifdef** to control compilation based on whether or not this symbol is defined.
- If used while linking, it links in *libnx.a*, the library that contains all the calls in the *Paragon System C Calls Reference Manual*. It also links in *libmach.a* and *options/autoinit.o*.

ICC *(cont.)*

ICC *(cont.)*

- If used while linking, it links in a special start-up routine that automatically copies the program onto multiple nodes, as specified by standard command line switches and environment variables. See the *Paragon System User's Guide* for information on these command line switches and environment variables.

-o*file* Uses *file* for the output file, instead of the default **a.out** (or *file*.o if used with the **-c** switch).

-O[*level*] Set the optimization level:

0	A basic block is generated for each C++ statement. No scheduling is done between statements. No global optimizations are performed.
1	Scheduling within extended basic blocks is performed. Some register allocation is performed. No global optimizations are performed.
2	All level 1 optimizations are performed. In addition, traditional scalar optimizations such as induction recognition and loop invariant motion are performed by the global optimizer.
3	All level 2 optimizations are performed. In addition, software pipelining is performed.
4	All level 3 optimizations are performed, but with more aggressive register allocation for software pipelined loops. In addition, code for pipelined loops is scheduled several ways, with the best way selected for the assembly file.

If a *level* is not supplied with **-O**, the optimization level is set to 2. If **-O** is not specified, the default level is 1. Setting optimization to levels higher than 0 may reduce the effectiveness of symbolic debuggers.

-P Preprocesses each file and leaves the output in a file named *file*.i for each file named *file*.c.

-r Generates a relinkable object file. (Passed to the linker.)

-rc Specifies the name of the driver configuration file.

-.suffix Saves the intermediate file in a file with the specified suffix when used with **-P**.

ICC *(cont.)*

ICC *(cont.)*

-s Strips symbol table information. (Passed to the linker.)

-S Skips the link and assemble step. Leaves the output from the compile step in a file named *file*.s for each file named *file*.c.

-show Displays the driver configuration parameters on startup.

-t *argument* Controls instantiation of template functions. *argument* can be one of the following:

- **all** — All template functions are instantiated.
- **none** — No template functions are instantiated.
- **local** — Only the functions used in the compilation are instantiated, and they are forced to be local. Note that this may cause multiple copies of local static variables. If this occurs, the program may not execute correctly.
- **used** — Only the functions used in the compilation are instantiated.

-time Print execution times for the compilation steps.

-U*name* Remove any initial definition of *name* in the preprocessor. (See also the **nostddef** option of the **-M** switch.)

The **-U** switch affects only *predefined* preprocessor macros, not macros defined in source files. The following macro names are predefined: **__cplusplus**, **__LINE__**, **__FILE__**, **__DATE__**, **__TIME__**, **__STDC__**, **__i860**, **__i860__**, **__PARAGON__**, **__OSF1__**, **__PGC__**, **_PGC_**, **_COFF**, **unix**, **MACH**, **CMU**, and **__NODE** (**__NODE** is only defined when compiling with **-nx**). Note that some of these macro names begin and/or end with *two* underscores.

Because all **-D** switches are processed before all **-U** switches, the **-U** switch overrides the **-D** switch.

-u *symbol* Initialize the symbol table with *symbol*, which is undefined for the linker. An undefined *symbol* triggers loading of the first member of an archive library.

-v Prints the entire command line for each tool as it is invoked, and invokes each tool in verbose mode (if it has one).

-V Prints the version banner for each tool (assembler, linker, etc.) as it is invoked.

-VV Displays the driver version number and the location of the online release notes. No compilation is performed.

ICC *(cont.)*

ICC *(cont.)*

-W*pass,option*[**,***option*...]
Passes the specified *option*s to the specified *pass*:

c	C++ front-end.
0 (zero)	C++ back-end.
a	Assembler.
l	Linker.
n	Symbol table lister.
m	Muncher.
p	Prelinker.

Each comma-delimited string is passed as a separate argument.

-w Do not print warning messages.

-X Generate cross-reference information and place output in the specified file.

-Y*pass,directory*
Looks for the specified *pass* in the specified *directory* (rather than in the default location), where *pass* is one of the following:

c	Search for the C++ front-end in *directory*.
0 (zero)	Search for the C++ back-end in *directory*.
a	Search for the assembler executable in *directory*.
l	Search for the linker executable in *directory*.
S	Search for the start-up object files in *directory*.
I	Set the compiler's standard include directory to *directory*.
L	Set the first directory in the linker's library search path to *directory* (passes **-YL***directory* to the linker).
U	Set the second directory in the linker's library search path to *directory* (passes **-YU***directory* to the linker).

ICC *(cont.)*

ICC *(cont.)*

See the **ld860** manual page for more information on the **-YL**, **-YU**, and **-YP** switches.

Description

iCC is the interface to the Paragon C++ compiler. It invokes the C++ compiler, assembler, linker, muncher, and prelinker with options derived from its command-line arguments.

iCC bases its processing on the suffixes of the files it is passed. Files whose names end with .cc, .c, .cpp, or .C, are considered to be C++ source files. They are preprocessed, compiled and assembled. The resulting object file is placed in the current directory. Files whose names end with .s are considered to be i860 assembly language files. They are assembled and the resulting object file is placed in the current directory. Files whose names end with .o are taken as object files, and are passed directly to the linker if linking is requested. Files whose names end with .a are taken as ar libraries. No action is performed on .a files unless linking is requested.

Files not ending in .cc, .cpp, .C, .c, .o, .s, or .a are taken as object files and passed to the linker (if linking is requested) with a warning message.

If a single C++ program is compiled and linked with one **iCC** command, then the intermediate object and assembly files are deleted.

Environment

iCC runs in two distinct environments. The first is a cross environment, where the compiler runs on one host and generates code for a different host. The second is a native environment, where the compiler and the generated code both run on the Paragon system.

Each of these environments use different directories for the executables, different libraries, and different default options. In addition, there may be custom installations that define their own default parameters to the compilers. The remainder of this section describes the implementation for the two standard environments, native and cross-compilation.

Native Environment

Executables reside in */usr/ccs/bin*; libraries and objects reside in */usr/ccs/lib*. The C compilation system (CCS) must be installed to use **iCC** on the Paragon system.

/usr/ccs/bin/icpp1	C++ compiler front end.
/usr/ccs/bin/icpp2	C++ compiler back end which reads the binary intermediate file produced by the front end and generates an assembly language file

ICC *(cont.)*

ICC *(cont.)*

/usr/ccs/bin/as	ELF assembler.
/usr/ccs/bin/ld	ELF linker
/usr/ccs/bin/iprelnk	C++ prelinker.
/usr/ccs/bin/imunch	C++ muncher.
/usr/ccs/bin/iCC	C compilation driver.
/usr/ccs/lib/libc.a	C runtime library.
/usr/ccs/lib/libC.a	C++ runtime library.
/usr/ccs/lib/libm.a	Math library.
/usr/ccs/lib/libstrm.a	Iostreams library.
/usr/ccs/lib/libcmplx.a	Complex library.
/usr/ccs/lib/ieee/libm.a	More accurate (and slower) math library.
/usr/ccs/bin/crt0.o	startup routine for the compilation environment.
/usr/ccs/bin/crti.o /usr/ccs/lib/values-Xa.o-	
/usr/ccs/lib/crtn.o	C startup/endup routines
/usr/ccs/lib/libic.a	C built-in intrinsic library.
/usr/ccs/lib/subchk.o	Array bounds checking routines.
/usr/ccs/lib, /usr/lib	Library search directories.
/usr/include	Include files for C library.

Cross Environment

Executables reside in $PARAGON_XDEV/bin.arch (where arch identifies the architecture of the system, e.g. sgi, solaris or sun4); libraries and objects reside in $PARAGON_XDEV/lib-coff. Include files are searched for in $PARAGON_XDEV/include/CC. PARAGON_XDEV is the root of the compiler installation directory.

$(PARAGON_XDEV)/paragon/bin.arch/icpp1 - C++ front-end compiler.

$(PARAGON_XDEV)/paragon/bin.arch/icpp2 - C++ back end.

ICC *(cont.)* ICC *(cont.)*

$(PARAGON_XDEV)/paragon/bin.arch/as860 - COFF assembler

$(PARAGON_XDEV)/paragon/bin.arch/ld860 - COFF linker

$(PARAGON_XDEV)/paragon/bin.arch/iprelnk - C++ prelinker

$(PARAGON_XDEV)/paragon/bin.arch/imunch - C++ muncher

$(PARAGON_XDEV)/paragon/bin.arch/iCC - C++ compilation driver

$(PARAGON_XDEV)/paragon/lib-coff/libm.a - Fast, less accurate, scalar math library

$(PARAGON_XDEV)/paragon/lib-coff/libstrm.a - C++ Iostreams library

$(PARAGON_XDEV)/paragon/lib-coff/libcmplx.a - C++ Complex library

$(PARAGON_XDEV)/paragon/lib-coff/libC.a - C++ runtime library

$(PARAGON_XDEV)/paragon/lib-coff/libc.a - C library

$(PARAGON_XDEV)/paragon/lib-coff/libb.a - Slow, more accurate, scalar math library

$(PARAGON_XDEV)/paragon/lib-coff/crt0.o - C startup routine

$(PARAGON_XDEV)/paragon/lib-coff/libic.a - C built-in intrinsic library

$(PARAGON_XDEV)/paragon/include/CC - Include files for C++ libraries

Files

a.out	executable output file.
file.a	library of object files.
file.c	C++ source file.
file.i	C++ source file after preprocessing.
file.o	object file.
file.s	assembler source file.
.iCCrc	defines the **iCC** driver's startup file. This file sets parameters for the driver's default configuration.

ICC *(cont.)*

ICC *(cont.)*

See Also

ar860, as860, dump860, if77, ifixlib, ld860, nm860, size860, strip860

Paragon System C++ Compiler User's Guide

LD860 LD860

ld860, ld: Link editor for Paragon™ system object files.

Cross-Development Syntax

ld860 [*switches*] *filename ...*

Native Syntax

ld [*switches*] *filename ...*

Arguments

filename The name of the object file or library.

You may specify the following *switches* in any order:

-B *integer* Specify the address to use for the base of the **.bss** section for all following object modules. This switch may be used multiple times, and affects only objects that appear after the switch in the command line.

-contig Force the **.data** section to follow the **.text** section. Overrides **-d**.

-d *integer* Specify the address at which the **.data** section is to be loaded. The default is 0x4010000.

-D Display the C++ **.debug** section.

-D *integer* Specify the length of the **.data** section to be *integer* bytes. The **.data** section is padded with zero to the specified length, which may not be less than the summed length derived from the object modules.

-e *symbol* Specify *symbol* as the entry-point. The default entry-point is `start`.

-f *filelist* Read in a list of files to be linked from file *filelist*. Names in the file can be separated by a comma, a space, a tab, or a linefeed. This switch may be used multiple times.

-k Start the **.text** and **.data** sections exactly at the addresses specified by the **-T** and **-d** switches (or at the defaults if the switches are not given) without performing the normal modifications to those addresses to make the file pageable.

LD860 *(cont.)*

LD860 *(cont.)*

-l*library*	Load the library **lib***library***.a**. The library is loaded from the first library directory in the library search path in which a file of that name is encountered.
-L	Display the C++ **.line** section.
-L*directory*	Add *directory* to the beginning of the library search path.
-m	Generate a link map (listing of modules and addresses).
-o *objfile*	Put the output object file in *objfile*. If this switch is not specified, the default object file name is *a.out*. If a file with the same name already exists, it is silently replaced.
-p	Align the **.data** section of the following module on a logical page boundary. (Other switches may appear between **-p** and the filename.) This switch may be repeated as necessary, and applies only to the next object file.
-P *integer*	Set the logical page size to *integer* bytes (default 65536). The value of *integer* must be a power of two multiple of 4096 bytes.
-r	Retain relocation entries in the output object file to allow incremental linking. The output object file produced with **-r** can be used as an input object file in another link. When **-r** is used, **-o** must also be specified.
-s	Strip all symbols from the output object file.
-t	Display the name of each object file or library as it is processed.
-T *integer*	Specify the address at which the **.text** section is to be loaded. The default is 0x10000. If used without **-d**, implies **-contig**.
-u *symbol*	Initialize the symbol table with *symbol*. The linker considers *symbol* to be undefined.
-V	Display the tool banner (tool name, version, etc.).
-y*file*	Load the library *file*. The library is loaded from the first library directory in the library search path in which a file of that name is encountered. (**-y** is like **-l**, but uses the specified filename without modifications.)
-YL*directory*	Replace the standard library directory (the first directory in the library search path) with *directory*.
-YU*directory*	Replace the secondary library directory (the second directory in the library search path) with *directory*.
-YP*directory*	Replace the entire library search path with *directory*.

LD860 *(cont.)* LD860 *(cont.)*

Description

Use **ld860** to link-edit the named file(s).

Object files and libraries are processed in the order specified.

Libraries are searched for unsatisfied externals when they are processed, and are not reopened to satisfy any symbols that might not have been satisfied. The search for libraries is done in the following order:

- If *PARAGON_LPATH* is defined, it is searched.
- If *PARAGON_LPATH* is not defined and *LPATH* is defined, it is searched.
- Any directories specified using the **-L** switch prior to **-l***libname* on the command line are searched.
- The standard default libraries are searched. In the cross-development environment, the default library directories are:

  $PARAGON_XDEV/paragon/lib-coff :*$PARAGON_XDEV/paragon/lib-coff/options*

  In the native environment, the default library directories are:

  $PARAGON_XDEV/usr/lib :*$PARAGON_XDEV/usr/lib/options*

  If *PARAGON_XDEV* is not set, */usr/lib:/usr/lib/options* is the default.

The search path used by the -l switch can be modified by any **-L**, **-YL**, **-YU**, or **-YP** switch to the left of the **-l** switch on the command line. The effect of these switches is cumulative.

The **-r** switch requires the **-o** switch.

If the **-r** and the **-s** switches are used together, the **-s** switch is ignored.

If the **-r** and the **-e** switches are used together, the **-e** switch is ignored.

If the **-f** switch is used, the **-B** and **-p** switches are applied as if the object file names appeared in place of the **-f** switch.

LD860 *(cont.)* LD860 *(cont.)*

The **-d** (data start address) and **-T** (text start address) switches interact as follows:

- If neither the **-d** nor the **-T** switch is used, the data and text start addresses default.
- If the **-d** switch is used without **-T** (that is, if a data start address is specified, but no text start address is specified), then the data start address specified is used, and the text start address defaults.
- If the **-T** switch is used without **-d** (that is, if a text start address is specified, but no data start address is specified), then the specified text start address is used, and the data section starts on the next logical page boundary following the end of the text section.
- If both the **-d** and **-T** switches are used, the specified data and text start addresses are used.

NOTE

Specifying addresses for the text and data sections different from the defaults may preclude the usage of profiling and performance monitoring tools. These tools require a gap between the text and data sections that is at least as long as the text section.

The profiling tools cannot be used on executables with a text section larger than 32 Mb, although such applications can be executed.

Special Symbols

The following symbols have special meanings to **ld860**:

_etext	The next available address after the end of the output section **.text**.
_edata	The next available address after the end of the output section **.data**.
_end	The next available address after the end of the output section **.bss**.

Programs should not use any of these as external symbols.

The symbols described above are those actually seen by **ld860**. Note that C and several other languages prepend an underscore (_) to external symbols defined by the programmer. This means that, for example, you cannot use **end** as an external symbol. If you use any of these names, you must limit its scope by using the **static** keyword in the declaration or declare the symbol to be local to the function in which it is used. If this is not possible, you will have to use another name.

LD860 *(cont.)*

LD860 *(cont.)*

See Also

ar860, as860, dump860, icc, if77, nm860, size860, strip860

MAC860 MAC860

mac860, **mac**: Macro preprocessor for the Paragon™ system.

Cross-Development Syntax

mac860 [*switches*] *sourcefile*

Native Syntax

mac [*switches*] *sourcefile*

Arguments

sourcefile Source file containing assembler and macro preprocessor commands.

You may specify the following *switches* in any order:

-D*sym=val* Defines *sym* as a local symbol with the value *val* in the macro preprocessor.

-I*incfile* Includes the file *incfile* before the first statement of *sourcefile*. You can use at most one **-I** switch in a single **mac860** command.

-o*objfile* Sets the output file name to *objfile* (the default is the name of the *sourcefile* with any *.s* suffix removed and *.mac* appended).

-V Displays the tool banner (tool name, version, etc.).

-y Makes the macro preprocessor output special directives that the assembler can use for better reporting of line numbers in the source file when errors are detected.

Description

The **mac860** command preprocesses the specified *sourcefile* with the macro preprocessor and produces a source file ready to be assembled with **as860**.

See Also

as860, **ar860**, **dump860**, **ld860**, **nm860**, **size860**, **strip860**

NM860 NM860

nm860, nm: Displays symbol table information for Paragon™ system object files.

Cross-Development Syntax

nm860 [*switches*] *filename ...*

Native Syntax

nm [*switches*] *filename ...*

Arguments

filename	The name of the object file or library.

You may specify the following *switches* in any order:

-d	Display numbers in decimal.
-e	Display external relocatable symbols only.
-f	Display all symbols, including redundant symbols. Overrides **-e**.
-h	Suppress headers.
-n	Sort symbols by name.
-o	Display numbers in octal.
-p	Use short form output. (See "Description" section.)
-r	Prepend the current file name to symbols.
-T	Truncate symbol names to 19 characters, plus an asterisk to indicate truncation.
-u	Display a list of undefined symbols.
-v	Sort symbols by value.
-V	Display the tool banner (tool name, version, etc.).
-x	Display numbers in hexadecimal (default).

NM860 *(cont.)* NM860 *(cont.)*

Description

Use **nm860** to display the symbol tables of the named file(s).

For each symbol in the output of the **-p** switch, one of the following characters identifies its type:

a	Absolute.
b	BSS section symbol.
c	Common symbol.
d	Data section symbol.
f	File tag.
r	Register symbol.
s	Other symbol.
t	Text section symbol.
u	Undefined.

In addition, the characters associated with local symbols appear in lowercase and the characters associated with external symbols appear in uppercase.

When using the **-v** or **-n** switches (sort by value or name, respectively), the scoping information is jumbled, so it is advisable to use the **-e** (externals only) switch.

See Also

as860, ar860, dump860, icc, if77, ld860, size860, strip860

SIZE860 SIZE860

size860, size: Displays section sizes of Paragon™ system object files.

Cross-Development Syntax

size860 [*switches*] *filenames*

Native Syntax

size [*switches*] *filenames*

Arguments

filename The name of the object file.

You may specify the following *switches* in any order:

-d Display sizes in decimal (default).

-f Full output.

-n Display the sizes of non-loading sections, as well.

-o Display sizes in octal.

-V Display the tool banner (tool name, version, etc.).

-x Display sizes in hexadecimal.

Description

Use **size860** to display the section sizes of the named files.

Note that the total size of an executable object may be greater than or less than the total of the sizes of all the compiled objects that make up the executable. This is because the true size of the BSS section is not known until after a set of objects is loaded, and because padding is done by **ld860** on other sections.

SIZE860 *(cont.)*

SIZE860 *(cont.)*

See Also

as860, ar860, dump860, icc, if77, ld860, nm860, strip860

STRIP860 STRIP860

strip860, strip: Strips symbol information from Paragon™ system object files.

Cross-Development Syntax

strip860 [*switches*] *filename ...*

Native Syntax

strip [*switches*] *filename ...*

Arguments

filename The name of the target object file.

You may specify the following *switches* in any order:

-l Strip line number information only.

-r Do not strip static, external, or relocation information.

-V Display the tool banner (tool name, version, etc.).

Description

Use **strip860** to strip symbol information from object files.

The default is to strip all symbols. This is generally only acceptable for executables.

See Also

as860, ar860, dump860, icc, if77, ld860, nm860, size860

Index

Symbols

.a extension
    Library filename 1-6

.C extension
    Source filename 1-6

.c extension
    Source filename 1-6

.cc extension
    Source filename 1-6

.cpp extension
    Source filename 1-6

.i extension
    Preprocessed file 1-7
    Preprocessed filename 1-6

.o extension
    Object file 1-7
    Object filename 1-6

.s extension
    Assembly-language file 1-7
    Assembly-language filename 1-6

__cplusplus
    usage 4-1

~complex 6-39

~filebuf 6-32

~fstream 6-21

~ifstream 6-22

~ios 6-9

~iostream 6-20

~iostream_withassign 6-24

~istream 6-14

~istream_withassign 6-24

~istrstream 6-25

~oftream 6-21

~ostream 6-18

~ostream_withassign 6-24

~ostrstream 6-26

~stdiostream 6-28

~streambuf 6-29

~strstream 6-27

~strstreambuf 6-34

A

a.out 1-3

abs 6-40

Aggregates
    Alignment 5-12, 5-14, 5-15, 5-17
    Data types 5-12

Alignment
    Aggregates 5-12

Arrays 5-14
Backward compatibility 5-12
Natural 5-12, 5-14, 5-15
Offset 5-14, 5-15
Padding 5-14, 5-15
Quad 5-14
Structures 5-14, 5-15
Unions 5-14

alignments of data types 5-10

ar manual page B-3

ar860 manual page B-3

arg 6-40

Arrays 5-12, 5-14

as manual page B-5

as860 assembler
manual page B-5

Assembler
Filenames 1-6
Inputs 1-3
Invocation 1-3
Outputs 1-3

Assembly-language file 1-7
Filename extension (.s) 1-6

attach 6-22, 6-32

B

bad 6-9

behavior, implementation-defined 5-10

bitalloc 6-10

Bit-field
Alignment 5-16

C

c assembling option
output 1-7

C driver 1-3

C switch (driver) 2-5

c switch (driver) 2-5

C++ extensions
#elif directive 5-9
#ident directive 5-10
#pragma directive 5-9
predefined macros 5-9

C++ language
extensions to 5-9

Class
Alignment 5-13

clear 6-10

close 6-23, 6-33

Compiler
Filenames 1-6
Inputs 1-3
Managing assembler and linker 1-3
Outputs 1-3

complex 6-38

conj 6-40

controlling the iCC driver 2-3

cos 6-40

cosh 6-40

cross-development environment 1-1

D

D switch (driver) 2-6

Data types
Array 5-12, 5-14
Class 5-12, 5-14
Fundamental 5-11
Scalar 5-11, 5-14
Structure 5-12, 5-14
Union 5-12, 5-14
void 5-17

data types, sizes and alignments of 5-10
dec 6-35
Definition
__cplusplus 4-1
development environments 1-1
driver
command lines, example 1-5
controlling 2-3
iCC v, 1-3, 2-1
icc 1-3
overview 1-3
driver switches
C 2-5
c 2-5
D 2-6
E 2-5
g 2-12
I 2-11
iCC (table) 2-1
K 2-14
L 2-14
l 2-14
lnx 1-4
M 2-7
m 2-14
nx 1-4, 2-16
O 2-12
o 2-16
P 2-5
r 2-13
S 2-5
s 2-13
U 2-6
V 2-16
v 2-16
VV 2-16
W 2-4
Y 2-4
dump860 manual page B-7

E

E switch (driver) 2-5
#elif directive 5-9
endl 6-37
ends 6-37
environment
execution 1-4
software development 1-1
eof 6-10
example driver command lines 1-5
execution environments 1-4
exp 6-40
extensions to C++ language 5-9

F

fail 6-10
fd 6-33
filebuf 6-32
Filename extension
.a (library file) 1-6
.C (source file) 1-6
.c (source file) 1-6
.cc (source file) 1-6
.cpp (source file) 1-6
.i (preprocessed file) 1-6
.o (object file) 1-6
.s (assembly-language file) 1-6
Input file 1-6
fill 6-10
flags 6-11
flush 6-18, 6-37
freeze 6-35
fstream 6-20
Function inlining 3-1

Fundamental
Data types 5-11

G

g switch (driver) 2-12

gcount 6-14

get 6-15

getline 6-15

getting started 1-1, 4-1, 6-1

good 6-11

H

hex 6-36

I

I switch (driver) 2-11

iCC driver v
controlling 2-3
invocation command 1-3, 2-1
switches (table) 2-1

icc driver 1-3

#ident directive 5-10

ifstream 6-22

ignore 6-16

imag 6-41

implementation-defined behavior 5-10

in_avail 6-29

#include, search rules for 5-10

Include file
Specifying 1-6

Inlining functions 3-1

Input file
Primary 1-6, 1-7

Input filename 1-6

Instantiation of templates 7-1

Inter-language calling
__cplusplus 4-1
C calling C++ 4-7
C++ calling C 4-7
C++ calling Fortran 4-9
Fortran calling C++ 4-8
Parameter passing 4-4
Return values 4-2
Underscore 4-3
Upper-lower case conventions 4-2

invoking
iCC driver 1-3, 2-1

io_state enum 6-8

ios 6-9

iostream 6-20

iostream_withassign 6-24

ipfx 6-16

is_open 6-33

istream 6-14

istream_withassign 6-24

istrstream 6-25

iword 6-11

K

K switch (driver) 2-14

L

L switch (driver) 2-14

l switch (driver) 2-14
ld manual page B-24
ld860 linker
  manual page B-24
libnx.a 1-4
Library
  Filename extension (.a) 1-6
  Linking 1-6
Linker
  Filename extensions 1-6
  Filenames 1-6
  Libraries 1-6
lnx switch (driver) 1-4
log 6-41

M

M switch (driver) 2-7
m switch (driver) 2-14
mac manual page B-29
mac860 manual page B-29
macros, predefined 5-9
Mangling 8-1
Manual pages B-1
manual, organization of v
Math header file
  Including 6-3
Math library
  Linking to 6-2

N

Name mangling 8-1
native development environment 1-1
nm manual page B-30
nm860 manual page B-30
norm 6-41
nx switch (driver) 1-4, 2-16

O

O switch (driver) 2-12
o switch (driver) 2-16
Object file 1-7
  Filename extension (.o) 1-6
oct 6-36
ofstream 6-21
open 6-23, 6-33
open_mode enum 6-8
opfx 6-18
organization of manual v
osfx 6-18
ostream 6-17
ostream_withassign 6-23
ostrstream 6-26
out_waiting 6-29
Output file
  Extension 1-7
  Temporary (work) files 1-3
overview
  driver (iCC) 1-3

P

P preprocessing option
  output 1-7
P switch (driver) 2-5
pcount 6-26
peek 6-16
polar 6-41
pow 6-41

#pragma directive 5-9

precision 6-11

Preprocessed file 1-7
    Filename extension (.i) 1-6

preprocessor macros, predefined 5-9

Primary input file 1-6, 1-7

put 6-18

putback 6-16

pword 6-11

R

r switch (driver) 2-13

rdbuf 6-12, 6-23, 6-25, 6-26

rdstate 6-12

read 6-16

real 6-42

resetiosflags 6-37

running a program
    on a single node 1-4
    on multiple nodes 1-4

S

S compiling option
    output 1-7

S switch (driver) 2-5

s switch (driver) 2-13

sbumpc 6-29

Scalars
    Data types 5-11

search rules for #include 5-10

seek_dir enum 6-8

seekg 6-17

seekoff 6-30

seekp 6-18

seekpos 6-30

setbuf 6-23, 6-29

setf 6-12

setfil 6-36

setiosflags 6-37

setprecision 6-36

setw 6-36

sgetc 6-30

sgetn 6-30

sin 6-42

sinh 6-42

size manual page B-32

size860 manual page B-32

sizes of data types 5-10

skip 6-12

snextc 6-30

software development environments 1-1

Source file
    Filename extenstion (.C) 1-6
    Filename extenstion (.cc) 1-6
    Filename extenstion (.cpp) 1-6

sputbackc 6-31

sputc 6-31

sputn 6-31

sqrt 6-42

stdiostream 6-28

stossc 6-31

str 6-27, 6-35

streambuf 6-28
strip manual page B-34
strip860 manual page B-34
strstream 6-27
strstreambuf 6-33
Structures
  Alignment 5-14, 5-15
  Size 5-14, 5-15
switches (driver)
  C 2-5
  c 2-5
  D 2-6
  E 2-5
  g 2-12
  I 2-11
  iCC (table) 2-1
  K 2-14
  L 2-14
  l 2-14
  lnx 1-4
  M 2-7
  m 2-14
  nx 1-4, 2-16
  O 2-12
  o 2-16
  P 2-5
  r 2-13
  S 2-5
  s 2-13
  U 2-6
  V 2-16
  v 2-16
  VV 2-16
  W 2-4
  Y 2-4
sync 6-17, 6-31
sync_with_stdio 6-12

T

tellg 6-17
tellp 6-19
Template instantiation 7-1
tie 6-13
types, sizes and alignments of 5-10

U

U switch (driver) 2-6
Unions 5-12, 5-14
unsetf 6-13

V

V switch (driver) 2-16
v switch (driver) 2-16
variables, sizes and alignments of 5-10
VV switch (driver) 2-16

W

W switch (driver) 2-4
width 6-13
write 6-19
ws 6-36

X

xalloc 6-13

Y

Y switch (driver) 2-4