CONVEX C Guide

First Edition

CONVEX

CONVEX COMPUTER CORPORATION

CONVEX C Guide

Order No. DSW-086

First Edition
April 1991

CONVEX Press
Richardson, Texas USA

CONVEX C Guide

Order No. DSW-086

© 1986, 1988, 1989, 1990, 1991 CONVEX Computer Corporation. All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, electronically stored, or reduced to machine readable form without prior written consent from CONVEX Computer Corporation.

Although the material contained herein has been carefully reviewed, CONVEX Computer Corporation does not warrant it to be free of errors or omissions. CONVEX reserves the right to make corrections, updates, revisions or changes to the information contained herein. CONVEX does not warrant the material described herein to be free of patent infringement.

UNLESS PROVIDED OTHERWISE IN WRITING WITH CONVEX COMPUTER CORPORATION (CONVEX), THE PROGRAM DESCRIBED HEREIN IS PROVIDED AS IS WITHOUT WARANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES. THE ABOVE EXCLUSION MAY NOT BE APPLICABLE TO ALL PURCHASERS BECAUSE WARRANTY RIGHTS CAN VARY FROM STATE TO STATE. IN NO EVENT WILL CONVEX BE LIABLE TO ANYONE FOR SPECIAL, COLLATERAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS, ARISING OUT OF THE USE OR INABILITY TO USE THIS PROGRAM. CONVEX WILL NOT BE LIABLE EVEN IF IT HAS BEEN NOTIFIED OF THE POSSIBILITY OF SUCH DAMAGE BY THE PURCHASER OF ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.

ConvexOS is a trademark of CONVEX Computer Corporation.

COVUE is a trademark of CONVEX Computer Corporation. COVUE products consist of COVUEbatch, COVUEbinary, COVUEedt, COVUElib, COVUEnet, and COVUEshell.

UNIX is a trademark of AT&T Bell Laboratories.

C1 is a registered trademark of CONVEX Computer Corporation.

CX/Motif is a trademark of CONVEX Computer Corporation.

C200 Series architecture is a trademark of CONVEX Computer Corporation.

VECLIB is a trademark of CONVEX Computer Corporation.

CRAY is a trademark of Cray Research, Inc.

IEEE is a trademark of the Institute of Electrical and Electronics Engineers, Inc.

Printed in the United States of America

Revision Information for

CONVEX C Guide

Edition	Document No.	Description
First Edition	720-000630-205	Released with CONVEX C software V4.1, April 1991. This document is a combination of documents released with the CONVEX C V4.0 compiler: *CONVEX ANSI C Concepts*, *CONVEX C Language Reference Manual*, and *CONVEX C User's Guide*. The origin of each chapter and appendix in this book is listed below: • Chapter 1, "Compiling a Simple Program," is from the *CONVEX C User's Guide.* • Chapter 2, "Compiler Fundamentals," is from the *CONVEX C User's Guide.* • Chapter 3, "Compatibility Modes," is new. Its contents were extracted from Chapters 2 and 3 and Appendixes A and B of *CONVEX ANSI C Concepts*. It also contains material from Chapter 1 of the *CONVEX C Language Reference Manual.* • Chapter 4, "Program Development Tools," is from the *CONVEX C User's Guide.* • Chapter 5, "Debugging Programs," is from the *CONVEX C User's Guide.* A short description of CXdb, the CONVEX Visual Debugger was added. • Chapter 6, "Profiling Programs," is from the *CONVEX C User's Guide.* • Chapter 7, "Optimizing C Programs," is from the *CONVEX C User's Guide.* • Chapter 8, "Mixed Language Programming," is from Chapter 3 of the *CONVEX C Language Reference Manual.* • Chapter 9, "CONVEX C Intrinsics," is a new chapter. • Chapter 10, "Input and Output," is from Chapter 4 of the *CONVEX C Language Reference Manual.* • Chapter 11, "Runtime Library," is from Chapter 5 of the *CONVEX C Language Reference Manual.* • Chapter 12 "The Preprocessor," is from Chapter 6 of the *CONVEX C Language Reference Manual.*

- Chapter 13, "The asm Statement," is from Chapter 7 of the *CONVEX C Language Reference Manual.*
- Appendix A, "Data Types and Representations," is from Chapter 2 of the *CONVEX C Language Reference Manual.*
- Appendix B, "Pragmas," is new. The `force_parallel_ext` and `prefer_parallel_ext` pragmas are new.
- Appendix C, "Compiler Directives," is new.
- Appendix D, "Cray Intrinsic Functions," is new.
- Appendix E, "Implementation-Defined Features," is from Chapter 4 of the *CONVEX ANSI C Concepts.*
- Appendix F, "C Manpages," is new.
- Appendix G, "Error Messages," is from the *CONVEX C User's Guide.* It has been updated.

Table of Contents

About This Guide ... xi
Organization ... xi
Notational Conventions ... xiii
Associated Documents ... xiv
Ordering Documentation ... xv
Technical Assistance ... xv

1 Compiling a Simple Program ... 1-1
What Is a Compiler? ... 1-1
Creating a Source File ... 1-1
Compiling One Source File ... 1-2
Compiling More than One Source File ... 1-3
Compiler and Linker ... 1-4
Libraries ... 1-5

2 Compiler Fundamentals ... 2-7
Introduction ... 2-7
Compiler Features ... 2-7
File-Naming Conventions ... 2-8
Command Line Format ... 2-8
Compilation Process ... 2-8
Compiler Command Line Options ... 2-9
Compatibility Options ... 2-9
Preprocessor Options ... 2-11
Code Generation Options ... 2-12
Diagnostic Options ... 2-15
Debugging and Profiling Options ... 2-15
Optimization Options ... 2-17
Miscellaneous Options ... 2-20
Compatibility with Options of Other Compilers ... 2-20
Predefined Symbols ... 2-20
Linker Use ... 2-22
Environment Variables ... 2-22
Compiler Messages ... 2-23
Mixed Compatibility Modes ... 2-24
Object File Compatibility ... 2-25

3 Compatibility Modes ... 3-27
Description of Four Modes ... 3-27
Specifying Compatibility Modes ... 3-28
Mode Conversion ... 3-30
Extended Mode Differences ... 3-33
Incompatibilities of Common C ... 3-38

4 Program Development Tools ... 4-43
lint Utility ... 4-43
make Utility ... 4-45
indent Utility ... 4-46
error Utility ... 4-47

5 **Debugging Programs** 5-49
Introduction 5-49
CXdb Debugger 5-49
CONVEX Consultant Debuggers 5-50
Cross-Reference Generator 5-52
Assembly-Language Debugger 5-53

6 **Profiling Programs** 6-55
Introduction 6-55
CONVEX Consultant Profilers 6-55
CXpa Profiler 6-57

7 **Optimizing C Programs** 7-59
Introduction 7-59
Scalar Optimization 7-60
Vector Optimization 7-63
Parallel Optimization 7-64
The Optimization Report 7-65
User-Directed Optimization 7-67

8 **Mixed-Language Programming** 8-69
C Function Calls 8-69
Accessing FORTRAN Routines from C 8-74
Accessing Ada Routines from C 8-78

9 **CONVEX C Intrinsics** 9-79
What Are Intrinsics? 9-79
Intrinsic Function Behavior 9-80
How to Disable Intrinsics 9-81

10 **Input and Output** 10-83
File Input and Output Concepts 10-83
Program Input and Output 10-86
Program Input and Output Example 10-86

11 **Runtime Library** 11-91
Functions Versus Function-like Macros 11-92
Calling Runtime Functions 11-93
assert.h 11-93
ctype.h 11-94
errno.h 11-95
float.h 11-96
limits.h 11-98
locale.h 11-100
math.h 11-101
setjmp.h 11-105
signal.h 11-106
stdarg.h 11-109
stddef.h 11-110
stdio.h 11-110
stdlib.h 11-117
string.h 11-121
time.h 11-124

12 Preprocessor Directives 12-127
#define 12-127
#include 12-129
#undef 12-129
Macro Operators 12-130
Conditional Compilation 12-131
#pragma 12-132
#error 12-132
#line 12-132

13 The asm Statement 13-133
Assembly-Language Statements 13-133

Appendixes

A Data Types and Representations A-135
Integral Types A-135
Floating-Point Types A-139
Pointer Data Type A-142
void Data Type A-143
union Data Type A-143
struct Data Type and Representation A-144
Array Data Type and Representation A-145

B Pragmas B-149
begin_tasks, next_task, end_tasks B-151
force_parallel B-151
force_parallel_ext B-152
force_vector B-152
max_trips B-153
no_recurrence B-153
no_side_effects B-154
no_parallel B-154
no_vector B-154
prefer_parallel B-155
prefer_parallel_ext B-155
prefer_vector B-155
pstrip B-155
scalar B-156
select B-157
synch_parallel B-158
unroll B-158
vstrip B-159

C Compiler Directives C-161
Compiler-Directive Syntax C-161

D Cray Intrinsic Functions D-163
Introduction D-163
Function Descriptions D-164

Example D-165

E Implementation-Defined Features E-167
Translation E-167
Environment E-167
Identifiers E-168
Characters E-168
Integers E-169
Floating-Point Numbers E-170
Arrays and Pointers E-170
Registers E-170
Structures, Unions, Enumerations, and Bit Fields E-170
Qualifiers E-171
Declarators E-171
Statements E-172
Preprocessing Directives E-172
Library Functions E-172

F C Manpages F-181
cc.1 .... F-182
cpp.1 .... F-198
lint.1 .... F-203
intro.3bit F-212
bint.h.3bit F-214
bitchange.3bit F-216
bitcount.3bit F-217
bitmask.3bit F-219
bitmit.3bit F-220
bitshift.3bit F-221

G Error Messages G-223
Error Message Control G-223
Error Message Catalog G-229

Index 325

List of Tables

2-1 File-Extension Conventions 2-8
2-2 Compatibility Modes 2-10
2-3 Option Compatibility 2-20
3-1 Compatibility Modes 3-27
3-2 Trigraph Representations 3-35
5-1 Postmortem Dump Contents 5-51
6-1 Compiler Options for Profiling 6-56
7-1 Optimization Pragmas 7-68
8-1 FORTRAN and C Declarations 8-75
A-1 Integral Type Bit Length A-143
A-2 Integral Ranges A-144
A-3 long long data type range A-145

A-4 Floating-Point Bit Length A-147
A-5 Native and IEEE Floating-Point Ranges A-147
A-6 Native and IEEE Floating-Point Ranges A-147
A-7 long float Range: Native and IEEE A-150
B-1 Restrictions on Pragma Use B-158
C-1 Compiler Directives C-170
E-1 Character Constant Representation E-176
E-2 Integer Conversion E-177
E-3 Characters Checked by ctype.h Functions E-181
E-4 Math Function Return Values E-181
E-5 Available Signals and their Semantics E-182
E-6 Default Actions for Signals E-183
E-7 errno values of fgetpos and ftell E-185
E-8 Error Messages E-186
E-9 Math Error Messages E-187
E-10 Nonblocking and Interrupt I/O Error Messages E-187
E-11 Ipc/Network Argument Error Messages E-187
E-12 NFS Error Messages E-187
E-13 SystemV Record Locking Error Messages E-188
E-14 Ipc/Network Operational Error Messages E-188
E-15 Tape System Error Messages E-188
F-1 Diagnostic Condition Default Settings F-196
F-2 Diagnostic Condition Default Settings F-197
11-1 Compatibility Modes 11-91
11-2 Math Function Return Values 11-104
11-3 errno values of fgetpos, fsetpos, and ftell 11-115
A-1 Integral Type Bit Length A-135
A-2 Integral Ranges A-136
A-3 long long data type range A-137
A-4 Floating-Point Bit Length A-139
A-5 Native and IEEE Floating-Point Ranges A-139
A-6 Native and IEEE Floating-Point Ranges A-139
A-7 Native and IEEE long float Range A-142
B-1 Restrictions on Pragma Use B-150
C-1 Compiler Directives C-161
E-1 Character Constant Representation E-168
E-2 Integer Conversions E-169
E-3 Characters Checked by ctype.h Functions E-173
E-4 Math Function Return Values E-173
E-5 Signals and Semantics E-174
E-6 Default Actions for Signals E-175
E-7 errno values of fgetpos and ftell E-177
E-8 Math Error Messages E-177
E-9 Nonblocking and Interrupt I/O Error Messages E-177
E-10 NFS Error Messages E-177
E-11 SystemV Record Locking Error Messages E-177
E-12 Error Messages E-178
E-13 Ipc/Network Argument Error Messages E-179
E-14 Ipc/Network Operational Error Messages E-179
E-15 Tape System Error Messages E-180
G-1 Diagnostic Condition Default Settings G-228
G-2 Diagnostic Condition Default Settings (cont) G-229

List of Figures

1-1 Role of the Compiler 1-1
1-2 Sample Program, prog2.c 1-3
1-3 Sample Program, file2.c 1-3
1-4 Compiling Multiple Source Files 1-4
1-5 Compiler and Linker Interactions 1-5
1-6 Linking Library Routines 1-6
2-1 Compilation Process 2-9
2-2 Options Required for Debuggers and Profilers 2-16
2-3 Object File Compatibility 2-25
8-1 Top of the Runtime Stack 8-70
8-2 Stack Layout 8-72
A-1 char Representation A-144
A-2 short int Representation A-144
A-3 int Representation A-145
A-4 long long Representation A-146
A-5 Single-Precision Floating Representation A-148
A-6 Double-Precision Floating Representation A-149
A-7 Pointer Representation A-150
A-8 Character Data Representation A-155
A-9 Character String Representation A-155
A-1 char Representation A-136
A-2 short int Representation A-136
A-3 int Representation A-137
A-4 long long Representation A-138
A-5 Single-Precision Floating Representation A-140
A-6 Double-Precision Floating Representation A-141
A-7 Pointer Representation A-142
A-8 Character Data Representation A-147
A-9 Character String Representation A-147

About This Guide

This document describes how to use the CONVEX C compiler, which is compatible with ANSI C and the POSIX operating system standard. This compiler is also backward-compatible with previous versions of CONVEX C. Chapters 1 through 7 are appropriate for novice programmers, while the remaining material is more appropriate for experienced programmers.

Organization

The first part of this guide is organized as follows:

- Chapter 1, "Compiling a Simple Program," provides an introduction to basic features of the CONVEX C compiler.
- Chapter 2, "Compiler Fundamentals," describes options that are available on the command line.
- Chapter 3, "Compatibility Modes," describes the four compatibility modes, how they are specified on the command line, how to convert an application to a particular mode, and differences among modes.
- Chapter 4, "Program Development Tools," describes tools that assist in program development, including the `lint`, `make`, `indent`, and `error` utilities.
- Chapter 5, "Debugging Programs," describes tools that aid in finding errors in a program.
- Chapter 6, "Profiling Programs," describes tools that are used to find time-consuming parts of a program.
- Chapter 7, "Optimizing C Programs," describes basic techniques that the compiler uses to optimize a program.
- Chapter 8, "Mixed Language Programming," describes the method that CONVEX C uses to call C functions and the interaction of CONVEX C with CONVEX FORTRAN and CONVEX Ada.
- Chapter 9, "CONVEX C Intrinsics," explains what intrinsic functions are, problems encountered using them, and how to correct those problems.
- Chapter 10, "Input and Output," discusses some of the functions that provide input and output for a program.
- Chapter 11, "Runtime Libraries," provides a brief summary of the functions declared in the ANSI C header files.

- Chapter 12, "Preprocessor Directives," describes the C preprocessor directives.

- Chapter 13, "The asm Statement," describes the syntax of the `asm` statement, a CONVEX extension that inserts assembly-language statements in a C program.

The appendixes in this guide are:

- Appendix A, "Data Types and Representations," lists CONVEX data types and their representations.

- Appendix B, "Pragmas," describes pragmas that the compiler recognizes.

- Appendix C, "Compiler Directives," describes the syntax that the obsolete compiler directives use.

- Appendix D, "Cray Intrinsics," describes Cray-compatible intrinsic functions that manipulate bits.

- Appendix E, "Implementation-Defined Features," lists the features of CONVEX C that may inhibit porting CONVEX C code to other ANSI C compilers.

- Appendix F, "C Manpages," contains man pages that are distributed with the CONVEX C compiler, including the cc(1), cpp(1), lint(1), intro(3bit), bint.h(3bit), bitchange(3bit), bitcount(3bit), bitmask(3bit), bitmil(3bit), and bitshift(3bit) man pages. Other man pages can be found in *ConvexOS Manpages for Programmers* (DSW-332).

- Appendix G, "Error Messages," describes the syntax and options of the `-d` compiler option and lists `cc` and `lint` error messages. Each error message explanation includes a short example and a corrective action.

Notational Conventions

The following conventions are used in this guide:

- The word "enter" in a phrase such as "enter a command" means that you type the command and press the carriage return key. In contrast, the word "type" (for example, "type a line of text") means that you do not press the carriage return key.
- *Italic* designates user-supplied variables in a command-line example, introduces important new terms, identifies variables in mathematical equations, and indicates titles of documents.
- `Constant-width font` is used for input and output. This includes: command names and options, functions, system calls, data structures and types, directives, pragmas, program statements, display examples, printout examples, and error messages returned.
- **Bold font** identifies user input in examples.
- Within command sequences:
  - Square brackets ([]) indicate optional input.
  - Curly brackets ({}) designate mandatory input, which must be one of two or more possible options. These options are separated by the pipe symbol (|).
  - Horizontal ellipsis (...) shows repetition of the preceding item(s).

  Consider the following example:

  COMMAND *input_file* [...]{a | b}[*output_file*]

  COMMAND must be typed as it appears; *input_file* indicates a file name that you must supply; the horizontal ellipsis in brackets indicates that additional input file names (either `a` or `b`) can be supplied; and *output_file* indicates an optional file name.
- References to man pages appear in the form `adb(1)`, where the name of the man page is followed by its section number enclosed in parentheses.
- Some constructs or coding practices may result in problems in the generated program. These are marked with a note or a caution box. A caution is considered more severe than a note. The note and caution boxes are included in the index.

Associated Documents

Using this software successfully can require information not specific to the tasks described herein or not within the scope of this guide.

The following documents are available in the C documentation set:

- *CONVEX C Optimization Guide* (DSW-089) presents a step-by-step method for program optimization. Background information is included and forms a foundation for concepts presented throughout the document.
- *CONVEX C Quick Reference* (DSW-087) provides quick access to function prototypes, compiler directives, compiler options, and language features.

The following is a partial list of other manuals or books that may provide more detailed information on ConvexOS and the topics presented in this manual:

- *ConvexOS Primer* (DSW-133) provides an introduction for users who have not previously used the ConvexOS operating system.
- *ConvexOS Manpages for Programmers* (DSW-332), is the standard reference for the ConvexOS function calls. This document contains an appendix that describes how to use the `contact` utility.
- *ConvexOS System Management Documentation Kit* (DSW-015) contains information needed to manage and administer a CONVEX supercomputer system.
- *ConvexOS Tutorial Papers* (DSW-170) is a collection of previously published papers that provides instruction in document preparation, programming, text editing, supporting tools and languages, system maintenance, and system implementation.
- *ConvexOS Utilities User's Guides Documentation Kit* (DSW-005) provides detailed information on the CONVEX Assembler, Linker, text editors, and `adb` debugger.
- *CONVEX CXpa User's Guide* (DSW-251) provides information required to use the CONVEX Performance Analyzer tool, an optional product.
- *CONVEX CXdb Concepts* explains debugging concepts that might be required to understand the CONVEX visual debugger.

For more information on the C language, refer to the following books:

- *American National Standard for Information Systems — Programming Language C.* Document Number: X3J11/90-013.
- *C: A Reference Manual,* by Samuel P. Harbison and Guy L. Steele, Jr., Prentice-Hall, Inc., 1987.
- *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall Inc., 1988.
- *IEEE Standard Portable Operating System Interface for Computer Environments,* IEEE Std 1003.1-1988.

Ordering Documentation

To order CONVEX documentation, complete the CONVEX Documentation and Subscription Service Order Form enclosed in the Documentation Catalog included with this manual.

To receive a specific edition of a manual, contact the local CONVEX sales office or call the Technical Assistance Center (TAC).

Technical Assistance

Hardware and software support can be obtained through the CONVEX Technical Assistance Center (TAC):

- From all locations in the continental U.S., call 1(800)952-0379.
- From locations in Alaska, Hawaii, and Canada, call 1(800)345-2384.
- From all other locations, contact the nearest CONVEX office.

Chapter 1
Compiling a Simple Program

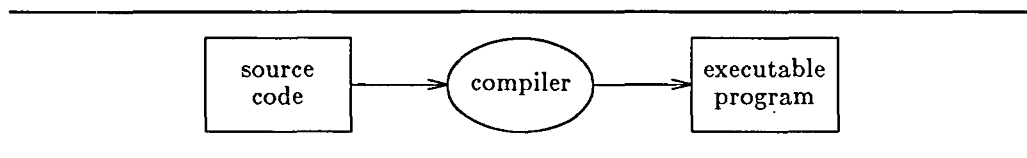
This chapter provides a brief introduction to compilation. It includes several small examples that show how to compile a program. It also defines some basic terms that are used frequently in this book. It is helpful if you understand how to program in C.

If you already understand how compilers, linkers, and libraries interact, skip this chapter. In this chapter, you must use a text editor to create or modify the source code for your programs; if you do not know how to edit files, refer to the *CONVEX Text Editors User's Guide* (DSW-010) for information about text editors.

What Is a Compiler?

Compilers are programs that translate other programs, usually from a high-level language, to machine or assembly language. The input to a compiler is source code for a program; the output is an executable program. Figure 1-1 illustrates the compiler's role:

Figure 1–1: Role of the Compiler



Creating a Source File

To create a C program to display the line "A simple program," use a text editor to create the code below:

```
#include <
```

Functions, also called subprograms or subroutines in other programming languages, are a basic building block in C. A function performs a small task in a program. The source file that you have just created contains one function definition, `main`. A C program contains only one `main` function. The `main` function in this source file calls the function `puts`, which displays the words contained in the double quotes.

Compiling One Source File

The name of the source file that you created has a "`.c`" extension. All files that contain source code for a C program must have this extension. (The COVUEshell environment requires "`.C`.") The compiler cannot compile a C language source file that does not have the "`.c`" extension.

To invoke the compiler enter

```
cc myprog.c
```

where cc is the command name of the compiler, and myprog.c is the name of the file to compile.

The compiler stores the executable program in a file called a.out. To run the resulting program, enter

```
a.out
```

"A simple program" is displayed. If a.out had already existed, its contents would have been overwritten.

The compiler has many optional features that you select by including options on the command line of the compiler. The format of an option consists of a hyphen followed immediately by the option name. Some options require arguments. The syntax for an option varies; some require a space to delimit the argument, while others do not. Refer to Chapter 2 for more information on command-line compiler options.

One option controls the name of the executable file. The format of this option is `-o filename`, where `filename` is the name of the executable file. For example, the following command-line creates an executable file named myprog:

```
cc myprog.c -o myprog
```

The `-D` option does not require a space to delimit its argument:

```
cc -DNDEBUG myprog.c -o myprog
```

This command line defines the macro `NDEBUG` with the source file myprog.c.

Compiling More than One Source File

Sometimes it is necessary to divide a program into multiple source files. For example, if the program is large, recompiling a piece of it takes less time than recompiling the entire program. Also, the functions of the program can be divided into source files containing related functions. For example, all input functions can be grouped in one file.

To understand how to create an executable program from several source files, create the two sample programs illustrated in Figures 1-2 and 1-3. Name these files prog2.c and file2.c, respectively.

Figure 1–2: Sample Program, prog2.c

```
#include <stdio.h>

extern void second_line( void );

main()
{
        (void) puts("The first line of my second program.");
        second_line();
}
```

Figure 1–3: Sample Program, file2.c

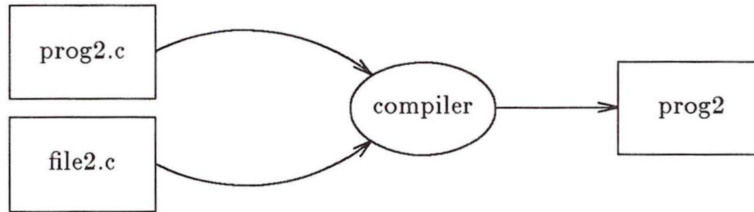
```
#include <stdio.h>

void second_line( void );

void second_line()
{
        (void) puts("The second line.");
}
```

These two files comprise one program in which two functions are defined: `main` is defined in prog2.c and `second_line` is defined in file2.c. The actions of the compiler are illustrated in Figure 1-4.

Figure 1–4: Compiling Multiple Source Files



The following command creates the executable program prog2:

```
cc prog2.c file2.c -o prog2
```

The compiler compiles the source files specified on the command line. After the executable program is created, run it by entering **prog2**.

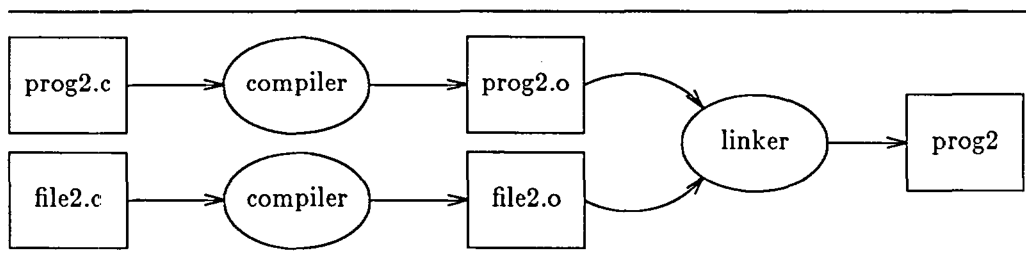
Compiler and Linker

The compiler does not create an executable program in one step. First, it creates an intermediate file that contains machine code; this intermediate file is called an object file. The object file has a "`.o`" file extension. For example, prog2.o is the object file of prog2.c, and file2.o is the object file of file2.c.

Most of the machine code is a translation of the source file, but some of it refers to source code in other files. The intermediate file obtained from the prog2.c source file created previously contains references to the source code in file2.c. Such references are called external

After the compiler has compiled all C source files listed on the command line into object files, it automatically invokes another program called a linker. The linker resolves the external references contained in the separate object files by combining the object files. When the object files are combined, there are no external references. The input to the linker is the object files generated by the compiler, and the output from the linker is the executable program. The process used to create the sample program prog2 is shown in Figure 1-5.

Figure 1–5: Compiler and Linker Interactions



When the `-c` option is used, the source file is compiled but not linked; this option interrupts the normal progress of the compiler. For example, the command line

```
cc -c file2.c
```

creates the object file file2.o, but not an executable file.

Similarly, it is possible to skip the compilation process and proceed directly to the linking process. For instance, if the object files prog2.o and file2.o already exist, the command line

```
cc prog2.o file2.o -o prog2
```

invokes the linker and creates the executable program prog2 because there are no source files to compile. If prog2.c is modified but file2.c is not, the following command line suffices:

```
cc prog2.c file2.o -o prog2
```

As another example, the command line

```
cc prog3.c file3.o file4.o -o prog3
```

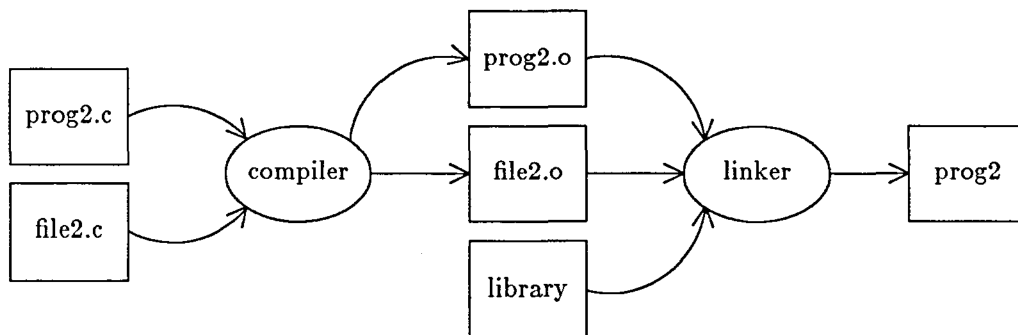
compiles prog3.c and then links prog3.o, file3.o, and file4.o.

Libraries

Libraries are composed of one or more object files. Some libraries contain functions that are defined to be part of the C language. Others might be collections of functions that you create for your programs. Each of the functions in the libraries can have external references. Like the external references in object files, the external references in libraries are not resolved until the linker is invoked.

For example, prog2.c calls a function called `puts`, which resides in a C library. After the source file is compiled, the object file contains a reference to the `puts` function. This reference is resolved by the linker. Figure 1-6 outlines when library routines are linked.

Figure 1–6: Linking Library Routines



The linker resolves external references in libraries and object files. When you invoke the cc command, the compiler automatically searches default libraries. Some command line options cause the compiler to search specific libraries. If you have developed your own libraries to link into an application, include them on the command line as you would an object file. For example, if mylib.a is the name of a library that contains functions called by a function in myprog.c, the following command line creates the executable program myprog:

```
cc myprog.c mylib.a -o myprog
```

The library mylib.a contains object files that the compiler uses to resolve function calls in myprog.c. Consequently, mylib.a appears after all other source and object files. The order in which you specify libraries on the command line can be significant. For example, if mylib.a precedes all source and object files on the command line, as in:

```
cc mylib.a myprog.c -o myprog
```

If myprog.c calls a function contained in the mylib.a library, no executable program will be created.

Chapter 2
Compiler Fundamentals

Introduction

The CONVEX C compiler translates a program written in C into an object file that can be combined with other object files and executed on a CONVEX computer. Previously compiled programs written in CONVEX C, CONVEX Assembly Language, or CONVEX FORTRAN can be linked with CONVEX C object code to produce an executable program.

This chapter discusses options that you can select on the command line of the CONVEX C compiler. The options are organized by function so that they are easy to locate. Some sections of related options contain comments that explain how they are used.

References are made to the Common C compiler. This compiler was formerly called the Portable C compiler and was delivered as part of ConvexOS. This compiler and previous releases of CONVEX C are no longer supported.

Compiler Features

The current release of the CONVEX C compiler conforms to the ANSI C standard, as specified in the ANSI X3J11/90-013 document. Programs written for the ANSI C conforming mode of the CONVEX C compiler can be compiled by other compilers with little or no modification to the source code. The converse is also true; conformance to ANSI C specifications increases portability of C programs across different computer systems.

While the new C compiler supports the ANSI C standard, it can also be extended with other environments. The compiler

- Is compatible with version 3.0 of CONVEX C
- Is compatible with most UNIX C compilers
- Contains extensions that customize code for CONVEX computers
- Complies with IEEE Standard 1003.1-1988 (POSIX)

The next two sections discuss file-naming conventions and compiler command line format.

File-Naming Conventions

Files specified to the CONVEX C compiler use the standard file extensions shown in Table 2-1.

Table 2-1: File-Extension Conventions

Files	File Extensions
C source files	.c
Compiled object files	.o
Symbolic assembly-language files	.s
Library files	.a

> **Note** If you are compiling under COVUEshell, the extension ".C" identifies C source files; for example, myfile.C is a C source file in the COVUE environment.

Command Line Format

The format of the C compiler command line is illustrated below:

cc [*options*] *files*

where

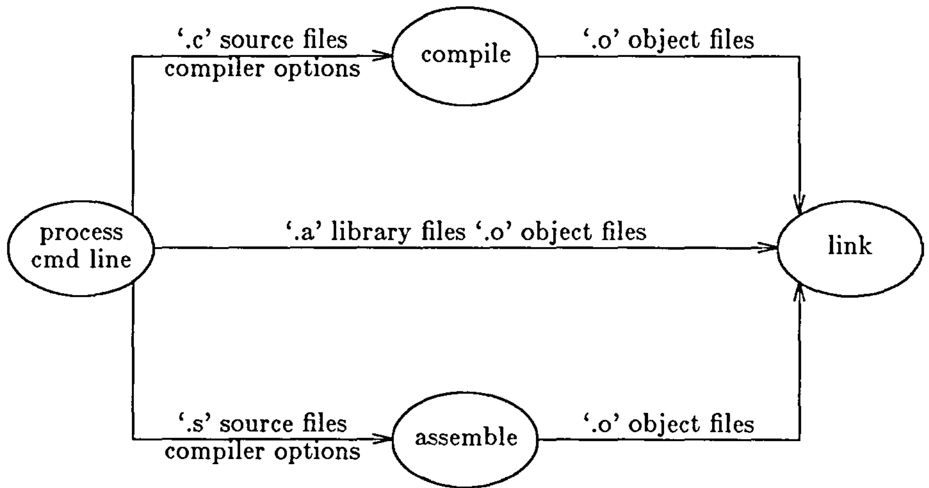
options is an optional argument specifying one or more of the options described in the rest of this chapter.

files represents one or more source files to be compiled, object files to link, assembly-language files to be assembled, or libraries to link.

Compilation Process

The compilation process is illustrated in Figure 2-1. The text that accompanies each arrow indicates the destination of the options and files that are entered on the command line.

Figure 2–1: Compilation Process



Options accepted by the assembler are common to the CONVEX C compiler and the CONVEX assembler. Descriptions of assembler options can be found in the *CONVEX Assembly Language User's Guide.*

Compiler Command Line Options

Compiler options can be categorized as follows:

- Compatibility options
- Preprocessor options
- Code generation options
- Diagnostic options
- Debugging and profiling options
- Optimization options
- Miscellaneous options

Each category is discussed in the remaining sections of this chapter.

Compatibility Options

The CONVEX C compiler can compile source code that conforms to the ANSI C standard, the POSIX standard, or both. The four compatibility modes are shown in Table 2-2:

Table 2-2: Compatibility Modes

Mode	Language	Default Functions
Extended (default)	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	ANSI C	ANSI C, POSIX
Strict	ANSI C	ANSI C
Backward-compatible	non-ANSI C	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first of the POSIX standards to be adopted. It was ratified on August 22, 1988, and represents a standard system call interface and environment based on the UNIX operating system. It is intended to support application portability at the source-code level.

The mode chosen depends on several factors:

- Portability of source code
- Existing source code
- Customization of source code for a system

For example, the backward-compatible mode is available for applications that were compiled by non-ANSI C compilers. The strict mode is used for programs that must be ported to other systems.

The options used to select the compatibility mode are as follows:

`-ext` Selects the extended mode. This mode provides an extended implementation of ANSI C and an ANSI C and POSIX compatible library system extensions. This is the default.

`-std` Selects the conforming mode. In this mode the compiler acts as a conforming ANSI C compiler. Only ANSI C and POSIX libraries are used.

`-str` Selects the strict mode. In this mode the compiler attempts to detect features of source files that prevent them from being strictly conforming ANSI C programs. Only ANSI C libraries are used.

`-pcc` Selects the backward-compatible mode. This is the mode most compatible with non-ANSI C compilers. Chapter 3, "Compatibility Modes," contains information required to convert an application compiled with the Common C compiler to the backward-compatible mode of CONVEX C.

Mode selection for the strict, extended, and backward-compatible modes is easy; just include the appropriate compiler option on the command line.

The following line compiles a program that conforms strictly with the ANSI C standard:

```
cc -str x.c
```

The following line creates a program for the extended mode (default):

```
cc file.c
```

In the extended mode, the _POSIX_SOURCE constant is defined automatically, permitting the use of POSIX functions.

To recompile an existing CONVEX program unchanged (backward-compatible mode), use the following command line:

```
cc -pcc file.c
```

The `-pcc` option is not completely compatible with the Common C compiler. Refer to Chapter 3, "Compatibility Modes," for information about the differences.

To use the conforming mode, define the _POSIX_SOURCE manifest constant on the command line using the `-D` compiler option or in the source files using the `#define` preprocessor directive.

To create a program that is POSIX conforming, define the constant _POSIX_SOURCE on the first line of each source file and compile with the following command line:

`cc -std` *files*

where *files* is replaced by the files that are used to create the program. This program could have been created using the `-D_POSIX_SOURCE` option on the command line instead of defining the `_POSIX_SOURCE` constant in the source files, but the program would not have been POSIX conforming.

Preprocessor Options

The preprocessor is the part of the compiler that translates the source file into tokens. Common uses of the preprocessor include the definition of the following:

- Macro constants
- Conditionally compiled source code
- Function-like macros

Several options are used to control the preprocessor. The preprocessor is also available as a standalone program. Refer to the cpp(1) and cc(1) man pages for additional information.

Options the compiler passes to the preprocessor are

-C	Prevents the preprocessor from removing comments.
-D*name*	Defines *name* with a default value of "1."
-D*name*=def	Defines *name* to the preprocessor, as if by the `#define` preprocessor directive.
-E	Runs only the C preprocessor on the named C source files and sends the result to standard output.

-I *dir* Names an alternate directory to search for include files. Several alternate directories can be specified; they are searched in the order specified on the command line. A maximum of eight -I options may be included on the command line.

-k Runs the preprocessor on the named C source files and generates dependency descriptions for **make**; results are printed out on the standard output stream. All file names on the command line that are not C source files are ignored.

-P Prevents the preprocessor from inserting control lines into the output of the preprocessor.

-U*name* Removes any initial preprocessor definition of *name*. The predefined identifiers of CONVEX C are described in a later section in this chapter.

The following example demonstrates how the -D option can be used effectively. Debugging code is included in a program to help locate errors. The preprocessor can be used to eliminate such code before it is compiled. For example, the following code is compiled only if the manifest constant DEBUG exists:

```
#ifdef DEBUG
printf("The file just written to is %s\n", filename );
#endif
```

To activate the debugging code, use this command line:

```
cc -DDEBUG prog.c
```

where prog.c is the source file that contains the example. Using this option is easier than defining the constant DEBUG within the source code itself.

For large programming projects with many source files and header files, relationships between the files might not be obvious. In such cases, the -k option is useful. This option creates a list of dependencies that can be used in a makefile. The format of files used by **make** is discussed in Chapter 4, "Program Development Tools."

Code Generation Options

The compiler includes some options that affect the code that is generated. For example, one option permits files to be compiled only; they are not linked after they are compiled. Other options control the format of floating-point numbers in the program. The code generation options are

-asm This flag is silently ignored; it is no longer required by code that contains **asm** statements.

-c Suppresses the linking phase of compilation. The object file that is generated from the file x.c or x.s is left in x.o.

`-extern` *arg*
: The `-extern` option can be used to control the interpretation of a program in which two different files declare an uninitialized variable (for example, `int i;`) at file scope. The argument *arg* can be

    `distinct`
    `same`

  When `same` is used, both files refer to the same variable. This is the default in all modes; it is traditional on BSD UNIX systems and is a conforming extension to Standard C. When `distinct` is used, the files refer to two distinct variables; the linker detects a multiply-defined symbol. This is the definition provided by the ANSI C standard. It is traditional on AT&T UNIX System V.

`-fd`
: A synonym for the `-float sp_ops` compiler option.

`-fi`
: Translate floating-point values into IEEE format and process in IEEE mode. This option requires that the machine be equipped with IEEE support hardware. If no floating-point format is specified, the site default is used.

`-float dp_const`
: Translate all unsuffixed floating-point constants into `double` data types. This is the default action of the compiler.

`-float sp_const`
: Translate all unsuffixed floating-point constants into `float` data types.

`-float dp_ops`
: Generate code to perform operations on 32-bit floating-point values using 64-bit instructions. The increase in precision can cause the program to run slower. This is the default in the backward-compatible mode.

`-float sp_ops`
: Generate code to perform operations on 32-bit floating-point values using 32-bit instructions rather than 64-bit instructions. The program can run faster, but results might not be as precise as those using 64-bit calculations. This is the default in the strict, conforming, and extended modes of CONVEX C.

`-fn`
: Translate floating-point values into native CONVEX format and process in native mode. If no floating-point format is specified, the site default is used.

`-parens` *arg*
: The `parens` option affects the interpretation of parentheses used in floating-point operations. The argument *arg* can be

    `explicit`
    `ignore`
    `implicit`

  If you specify the `explicit` argument, parentheses are honored regardless of the compatibility mode. If you specify the `ignore` argument, parentheses are ignored; the compiler can reorder any floating-point expression according to rules of associativity and commutativity. This is the default in the backward-compatible and extended modes. If you specify the `implicit` argument, the compiler honors all explicit parentheses, as well as those implied by grammatical precedence and associativity rules; no reordering can be performed. This is the default for the strict and conforming modes.

-re
: Generate reentrant code by creating both a scalar and parallel version of parallel loops. The default, at optimization level -O3, generates nonreentrant code for functions with parallel regions. This option has no effect on functions without parallel code. Use this option to compile a function called by parallelized regions of code. It is always safe to use -re; however, the text segment can be unnecessarily large if -re is used when it is not required.

-S
: Generate assembly code for each program unit in a source file. Assembler output for source file x.c is put in file x.s; no object file is generated and the linker is not invoked.

-string read_only
: Do not allow the program to modify string literals. This is the default in the strict, conforming, and extended compatibility modes of CONVEX C.

-string read_write
: String constants may be modified. This is the default in the backward-compatible mode of CONVEX C.

-tm *x*
: Specifies the target machine architecture. *x* can be c1, C1, c2, C2; the default value is the type of machine hosting the compiler. The c1 and C1 options indicate that a C1 series computer is the target architecture. The c2 and C2 options indicate that a C2 series computer is the target architecture. When invoking the linker, machine-dependent versions of some libraries are used.

The -tm option is useful, for example, when the program is being compiled for a C2 computer on a C1 computer. The command line

```
cc -tm C2 prog.c
```

compiles a version of program prog.c for use on a C2 computer. The resulting program will not run on a C1 computer.

Note

Because the cc command line places the object file of x.c in x.o and the object file of the assembly-language source file x.s in x.o, source files for both assembly language and C must not have the same file name stem (file name without the extension).

Diagnostic Options

These options produce additional information that can be used to judge the quality of the source file. Different options can be selected for varying levels of detail.

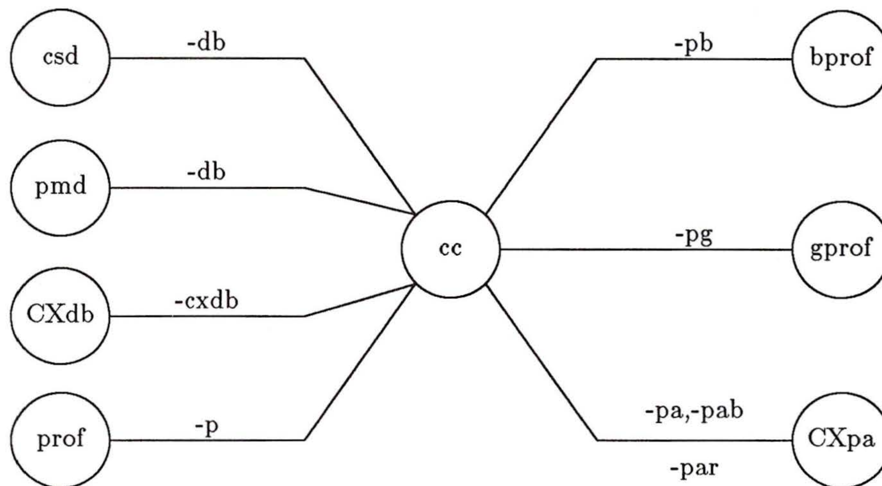
`-d` *name*[={w|e}]	Control issuance and severity of messages represented by *name*. This option can be used to convert a warning message to an error message and vice versa. Warning and error messages are listed in Appendix F, "Error Messages." If *name* is followed by `=w`, it is converted to a warning message. If *name* is followed by `=e`, it is converted to an error message. If *name* is not followed by `=w` or `=e`, the diagnostic message associated with *name* is suppressed.
`-or` *table*	Specifies contents of the optimization report to be produced. The value for *table* can be `none`, `all`, `array`, or `loop`. If this option is not specified, only the loop table is displayed. For additional information on this option, refer to the *CONVEX C Optimization Guide*.
`-sc`	Instructs the compiler to check the program for errors without performing optimization, vectorization, or code generation.
`-nw`	Suppresses all warning messages.

The `-sc` option is useful during the initial development of a program. This option reduces the load on a system by checking only for syntax errors.

Debugging and Profiling Options

Several debuggers and profilers are available. Each of these tools requires the inclusion of extra information, called instrumentation, in the compiled code to permit suitable interpretation of the program by the support tools. Figure 2-2 shows the options required to use each tool on an executable program.

Figure 2–2: Options Required for Debuggers and Profilers



The options include:

`-cxdb` Produces additional information for use by the CONVEX Visual Debugger, **cxdb**. This optional product is a symbolic debugger that can debug optimized code. Its window interface makes interacting with the debugger easier than interacting with line-oriented debuggers. Refer to the *CONVEX CXdb User's Guide* for more information.

`-db` Produces additional information for use by the symbolic debugger, `csd`, and the postmortem dump utility, `pmd`. It also passes the `-lg` option to the linker. If this option is used with any level of optimization, the statements in the source code can be reordered by the compiler. At optimization level `-no`, no source code reordering takes place. Further, variables in inner blocks are not known to the debugger if any optimization is performed.

`-p` Produces code that counts the number of times each routine is called. Profiled versions of system libraries are used instead of default libraries. If linking takes place, this option replaces the standard startup routine with one that automatically calls `monitor`(3) at the start and arranges to write out the file mon.out when the object program terminates. You can then generate an execution profile using `prof`.

`-pb` Includes instrumentation that produces an execution profile named bmon.out at normal termination. You can then obtain listings of source-level execution counts using `bprof`.

`-pg` Includes instrumentation in the manner of `-p`, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a file named gmon.out at normal termination. You can then generate an execution profile using `gprof`.

-pa Produces counting code for routine-level and loop-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

-pab Produces counting code for block-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

-par Includes instrumentation for routine-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

Optimization Options

Several compiler options affect optimization. For a general discussion of optimization, refer to Chapter 7, "Optimizing C Programs." For more specific information, such as optimizing a program using optimization directives, refer to the *CONVEX C Optimization Guide.*

It may be possible to use all of the optimization options. There are two versions of CONVEX C: a scalar optimizing version bundled with the operating system and a vectorizing/parallelizing version, which is an optional product. The scalar optimizing version of the compiler cannot do vector and parallel optimizations; the scalar compiler also ignores optimizing compiler options it does not contain.

-alias array_args Assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but can allow greater optimization to occur, particularly vectorization. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to the elements of the formal parameter (for example, `formalParameter[10] = x[10]`).

-alias ptr_args Assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). *The compiler generates incorrect code if this assumption is not true.* This conflicts with the language definition, but allows greater optimization to occur, particularly vectorization. This option cannot be used if the formal parameter itself is assigned by the function (for example, `formalParameter = &x[10]`); assignments can be made to variables identified by the formal parameter (for example, `formalParameter[10] = x[10]`).

-alias cautious Tries to compile with `-alias standard`, but if inappropriate constructs are found, compiles with `-alias worst` instead. This is the default in the ANSI C-compatible modes.

`-alias standard`
: Performs aliasing based on assumptions permitted in ANSI C; pointers of one object type can only reference objects of that same type. For example, a pointer to an object of type `float` cannot reference an object of type `int`. Such assumptions permit additional optimizations. If this option is specified when such conditions do not exist, the resulting program cannot function correctly.

`-alias worst`
: Performs pointer aliasing based on the assumption that pointers can modify objects of any type. This is the default in the backward-compatible mode.

`-ds`
: Causes the compiler to perform dynamic selection on loops the compiler selects on the basis of profitability. Multiple copies of the loop running sequential, vector, or parallel modes are generated; the optimal version is executed based on the trip count of the loop. This option is effective only at levels `-O2` and `-O3`.

`-ep` *n*
: Optimizes the code for *n* processors, but does not prevent the code from running on other system configurations. Single-program performance increases at the expense of system throughput. *n* must be in the range 1 through 4. It is effective only when you specify the `-O3` option. Refer to the *CONVEX C Optimization Guide* for additional information on using this option.

`-no`
: No machine-independent optimization is performed; this is the default, but you can override it with the `-O` options.

`-O`
: Performs the highest scalar optimization level available, `-O1`.

`-O`*n*
: Performs machine-independent optimizations, level *n*, where:
  - *n*=0 selects basic block scalar optimization
  - *n*=1 selects `-O0` plus global scalar optimization
  - *n*=2 selects `-O1` plus vectorization
  - *n*=3 selects `-O2` plus parallelization

  If you do not specify this option, the compiler performs no machine-independent optimization. The `-O3` option is not available on the C1. The `-O3` option can degrade performance if the program is executed on a single CPU.

`-rl`
: This option is equivalent to `-ds -ur`.

`-uo`
: Performs potentially unsafe optimizations. For example, the `-uo` can move the evaluation of common subexpressions and invariant code from within conditionally executed code. The code is then executed unconditionally.

`-ur`
: Causes the compiler to perform loop unrolling on loops the compiler selects on the basis of profitability. This option is effective at optimization levels `-O2` and `-O3`.

`-va`
: A synonym for `-alias array_args`

The following source code illustrates a situation in which the `-alias array_args` option can be used:

```
void vaf( char [] );
char global = 'A';

int main()
{
     char b[10];

     vaf( b );
     return(1);
}

void vaf( char array[] )
{
     int i;

     for( i=0; i<10; i++ )
          array[i] += global;
}
```

The loop in the vaf function is vectorized when the `-alias array_args` option is specified because the compiler can assume that the address of `global` is not the address of an element of `array`. Refer to the *CONVEX C Optimization Guide* for more information on this compiler option.

The following command line optimizes the C program file.c for use on a C240 machine.

```
cc -O3 -ep 4 -tm C2 file.c
```

The executable program created by this command line takes advantage of the instruction set and the availability of four processors of the C240 machine to decrease the completion time of the program.

The command line

```
cc -alias worst file.c
```

creates an executable program that uses the worst-case aliasing algorithm for compiling a program. The default aliasing algorithm for ANSI C assumes that pointers of one type do not point to objects of another type. For example, `int` pointers cannot point to objects with type `float`. For more information on this option, refer to the *CONVEX C Optimization Guide.*

Miscellaneous Options

`-B`*dir* Finds a substitute compiler in the directory *dir*. If *dir* is not specified, the backup compiler in the directory /usr/lib/oldcc is used. For example, the command `/usr/new/cc -B/usr/new` invokes `/usr/new/cocc` instead of the default `/usr/lib/oldcc/cocc`.

`-o` *name* Specifies that *name* is the name of the executable file produced by `ld`. If this option is not specified, the default name is a.out. If `-c` is specified and there is only one file to compile or assemble, *name* is the name of the object file produced.

`-tl` *time* Sets the maximum CPU time limit on compilations to *time* minutes. If the CPU exceeds the allotted time, the compilation is aborted.

`-vn` Identifies the compiler version. Outputs the version numbers of `cc`, `cpp`, `cocc`, and `errmsgc` to stderr.

Compatibility with Options of Other Compilers

To improve portability of `make` files and to enhance compatibility with C compilers supported by other compiler vendors, CONVEX C recognizes several additional compiler options. Table 2-3 lists these options and tells how the CONVEX C compiler interprets them.

Table 2-3: Option Compatibility

Option	Interpretation
`-g`	A synonym for the `-db` option
`-n`	Ignored with a warning
`-O`	A synonym for the `-O1` option
`-OL`	A synonym for the `-O1` option
`-V`	A synonym for the `-vn` option
`-w`	A synonym for the `-nw` option

Predefined Symbols

The following macros are predefined when the C compiler invokes the C preprocessor:

__convex__ This symbol is defined in all compatibility modes.

convex This symbol is defined in the backward-compatible mode.

_CONVEX_FLOAT_ The compiler defines _CONVEX_FLOAT_ when the compiler operates in native floating-point mode. Refer to the description of the `-fi` and `-fn` options in the "Code Generation Options" section of this chapter, for more information.

_CONVEX_SOURCE	This symbol is predefined in extended mode. In the conforming mode (`-std`) you will usually want to define the symbol _POSIX_SOURCE to make the POSIX symbols available in the include files.
__convexc__	This symbol is always defined. It is used to identify the compiler. Other symbols defined by the preprocessor (refer to the cpp(1) man page) identify the machine and operating system.
convexvc	This symbol is defined in the backward-compatible mode. Its use is obsolete; use the symbol __convexc__ instead.
__DATE__	The date of translation of the source file. It is a character string literal of the form "mmm dd yyyy". This is not available in the backward-compatible mode.
__FILE__	The name of the source file being compiled. This is predefined in all modes. You can modify this macro with the `#line` preprocessor directive, but not with the `#define` preprocessor directive or `-D` command-line option.
_IEEE_FLOAT_	The compiler defines _IEEE_FLOAT_ when the compiler operates in IEEE mode. Refer to the description of the `-fi` and `-fn` options in the "Code Generation Options" section of this chapter, for more information.
__LINE__	The line number of the current source line. This is predefined in all modes. You can modify this macro with the `#line` preprocessor directive, but not with the `#define` preprocessor directive or `-D` command-line option.
_POSIX_SOURCE	This symbol is predefined in extended mode. In the conforming mode (`-std`) you will usually want to define the symbol _POSIX_SOURCE to make the POSIX symbols available in the include files.
__stdc__	This symbol is defined when you compile in an ANSI C mode (`-str`, `-std`, or `-ext`). It indicates that the compiler is an ANSI C style compiler, but that it is not conforming. The conforming modes define both `__STDC__` and `__stdc__`.
__STDC__	The symbol __STDC__ is defined when either the `-std` or `-str` options are specified. The definition of this symbol indicates that the compiler conforms to the ANSI C standard. You cannot remove this definition with the `-U` option or `#undef`.
__TIME__	The time of translation of the source file. It is a character string literal in the form "hh:mm:ss". This is not available in the backward-compatible mode.
__unix__	This symbol is defined in all compatibility modes.
unix	This symbol is defined in the backward-compatible mode.

The compiler can predefine other names beginning with "__". Such names are reserved to CONVEX; their use or availability can change in subsequent releases. Avoid writing applications that depend on the *presence* or *absence* of names beginning with "__" (except those defined above).

Linker Use

The `cc` command invokes the linker directly. Several compiler options translate into linker options. You can also include additional linker options on the `cc` command line.

CONVEX recommends that you always use the appropriate compiler to invoke the linker rather than invoking it directly. This insulates the program from changes in library structure when new compiler releases are installed.

The compiler always passes the flags `-X`, `-NL`, and `-L/usr/lib` to the linker. The compiler flags `-fi` and `-fn` are passed to the linker. `-Eposix` is passed to the linker except when `-pcc` is given, in which case `-Enoposix` is passed.

The compiler compatibility mode controls the libraries searched by the loader; you normally control this by passing one or more `-l` options to the linker.

The following options, when present on the `cc` command line are passed to `ld` (refer to the ld(1) man page):

```
-A -F -L -M -T -X -e -l -m -r -s -t -u -x -y
```

Refer to the *CONVEX Loader User's Guide* for their meaning and use. For the `-l` option to be effective, you must specify it after all object files on the `cc` command line. Linker options that require a value must be written with no spaces between the flag and value (for example, `-L/mydir`) when passed through `cc`.

The `-link` option causes following arguments to be passed to `ld`. If an argument does not start with - or starts with `-l`, the argument is added to the linker's file list; otherwise it is added to the linker's flag list.

Do not use `-lc` and `-pcc` on the same `cc` command line; they are incompatible. Also, you do not have to include the `-lm` linker option on the `cc` command line.

Environment Variables

You can use the ConvexOS environment variables CCOPTIONS and CCLIBS to preset any of the CONVEX C compiler options. During compilation, the compiler processes options specified by these variables before processing options specified on the command line. If there is a conflict, the command line options take highest priority, and options specified with CCOPTIONS take least priority. These environment variables replace the VCOPTIONS variable used in previous releases of CONVEX C.

For example, the following command sets the optimization level at `-O2` and states that formal array parameters are to be treated as arrays rather than as pointers for all C programs that are compiled.

```
% setenv CCOPTIONS "-O2 -va"
```

With the previous value of CCOPTIONS, the following command compiles the file my_prog.c with an optimization level of -O1 and causes a warning message to appear.

```
% cc -O1 my_prog.c
Contradictory optimization level specifications given - believed '-O1'
```

Compiler Messages

This section describes messages the compiler displays during compilation or runtime. These messages are grouped into the following categories:

- Diagnostic messages
- Optimization report
- Runtime messages produced by the backward-compatible mode

Diagnostic Messages

If the compiler detects an error or a condition that requires an advisory, a suitable message is directed to standard error (stderr). You can redirect such messages to a file of your choice by using the standard shell redirection commands as described in *ConvexOS Manpages for Users* (refer to the csh(1) and sh(1) man pages). A compiler diagnostic message consists of the following components:

- Compiler name, cc
- Line number where the error occurred
- Path name of the source file containing the line in error
- A brief description of the error

Example:

If the compiler detects an internal error, it generates a message in the following format:

```
>>>>>  C O M P I L E R   E R R O R  <<<<<
>>>>> See your system manager for help <<<<<
cc : Compiler error on line num of filename.
```

The preceding lines are followed by a line that describes the nature of the error. Report such errors using the contact utility.

Optimization Report

If a program is compiled with the -O2 or -O3 option, the compiler generates an optimization report for each program unit. This report consists of a loop table, an array table, or both. You can specify the tables to be included in the optimization report with the -or compiler option. Refer to the *CONVEX C Optimization Guide* for more information on this output option.

Runtime Messages

Runtime error messages in the backward-compatible mode are directed to stderr. A math-routine error message has the form

```
routine_name: [error_number] description
```

Example:

```
mth$r_sqrt: [300] square root undefined for negative values
```

Mixed Compatibility Modes

The CONVEX C compiler supports four compatibility modes: extended, conforming, strict, and backward-compatible. Each of these modes is accessed using methods described in the "Compatibility Options" section of this chapter. However, it is possible to create a program that is compatible with a mixture of these modes. A two-step compilation process is necessary to obtain mixed mode programs. For example, to compile a program that has strict ANSI C language features but also accesses the POSIX function library, use the following command lines:

```
cc -c -str -D_POSIX_SOURCE file.c
cc -std file.o
```

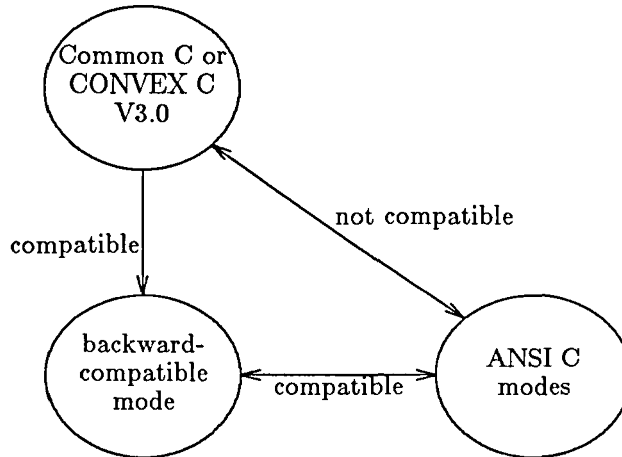
The first command line requires the file to be ANSI C compatible and to include POSIX header file information. The second command links the resulting object file with the default libraries.

For more information on the subject of mixed compatibility modes, refer to Chapter 3, "Compatibility Modes."

Object File Compatibility

You can continue to use object files and libraries created with CONVEX C V3.0 and the Common C compiler with CONVEX C. Figure 2-3 shows the relationships between those object files.

Figure 2–3: Object File Compatibility



The arrows in Figure 2-3 indicate the direction in which object files are compatible. For example, you can link object files created by the Common C compiler into applications that are compiled in the backward-compatible mode of CONVEX C. But you cannot link object files created in either the ANSI C modes or the backward-compatible mode of CONVEX C into applications compiled with either the Common C compiler or CONVEX C V3.0.

Chapter 3
Compatibility Modes

This chapter describes:

- Compatibility modes of the compiler
- How to specify each compatibility mode on the command line
- How to convert an application to a compatibility mode
- Differences between the compatibility modes

Description of Four Modes

CONVEX C has four compatibility modes, as shown in Table 3-1.

Table 3-1: Compatibility Modes

Mode	Language	Default Functions
Extended	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	ANSI C	ANSI C, POSIX
Strict	ANSI C	ANSI C
Backward-compatible	non-ANSI C, CONVEX	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first POSIX standard to be adopted. It was ratified on August 22, 1988, and represents a standard system call interface and environment based on the UNIX operating system. It supports application portability at the source-code level.

The four compatibility modes differ in two areas: language specification and library functions. For example, an application that is compiled in the strict mode uses only ANSI C language features and the only library functions that are automatically linked are those in the ANSI C library.

One language feature that can be used in the extended mode but not in the strict or conforming mode is the `asm` statement. This is a CONVEX extension that directs the compiler to insert in-line assembly-language statements into object code. Another CONVEX extension that is not permitted in either the conforming or strict modes is the data type `long long int`. This data type is a 64-bit integer. The extended mode is the default compatibility mode of the compiler. Some portability is sacrificed when CONVEX-specific language features and library functions are used.

The major difference between the conforming mode and the strict mode is the functions that can be automatically linked into the application program: the conforming mode can link in POSIX functions while the strict mode can only link in ANSI C functions. These two modes use the same language specification because POSIX defines an interface to system functions.

The backward-compatible mode differs considerably from the other three modes with respect to language features and functions. It does not recognize the keywords `const`, `signed`, and `volatile`. Further, it does not use ANSI C or POSIX functions. The only benefit of using this mode is that it is closely compatible with CONVEX C compilers prior to CONVEX C V4.0.

Specifying Compatibility Modes

Specifying a compatibility mode with the compiler is straightforward; the suitable compiler option is included on the command line. The following command-line examples show how to compile the source file applic.c in each of the four modes:

extended	`cc applic.c`
conforming	`cc -std -D_POSIX_SOURCE applic.c`
strict	`cc -str applic.c`
backward-compatible	`cc -pcc applic.c`

The macro constant _POSIX_SOURCE defined on the command line for the conforming mode provides access to POSIX function prototypes in ANSI C header files. Applications that conform to the POSIX standard must define the `_POSIX_SOURCE` macro on the first line of every source file. Applications that do not need to conform to the POSIX standard can define it on the command line, as in the example. This constant is not automatically defined in the conforming mode because conforming POSIX applications *must* define it in the source code.

Another macro constant that has a similar purpose is _CONVEX_SOURCE. This macro provides access to the CONVEX function prototypes in the ANSI C header files. Both _POSIX_SOURCE and _CONVEX_SOURCE are automatically defined when a source file is compiled in the default mode. The _CONVEX_SOURCE macro can be used *only* if the _POSIX_SOURCE macro is also defined.

Single-Mode Compilation Examples

Examples contained in this section demonstrate how to compile a program for each of the four modes of the compiler. For all the examples, assume that the `CCOPTIONS` environment variable is null. The following command line compiles an application for the extended mode, which is the default compatibility mode:

```
cc applic.c
```

Because _CONVEX_SOURCE and _POSIX_SOURCE are defined automatically, function prototypes for CONVEX and POSIX functions are accessible in the ANSI C header files.

The following command line compiles an application in the conforming mode:

```
cc -D_POSIX_SOURCE -std applic.c
```

No CONVEX extensions are used in this example; only ANSI C language features are permitted. The `-std` compiler option indicates that only the POSIX and ANSI C system functions are automatically linked into the executable program.

The following command line compiles an application in the strict mode that conforms to the ANSI C specification:

```
cc -str applic.c
```

The following command line compiles a program in the backward-compatible mode of the compiler:

```
cc -pcc applic.c
```

Use of either the _POSIX_SOURCE macro or the _CONVEX_SOURCE macro in the backward-compatible mode is undefined because these macros provide access to constructs that the backward-compatible modes does not recognize. Several compiler options are incompatible with this mode. Refer to Chapter 2, "Compiler Fundamentals," for more information.

Mixed Compatibility Modes

Each of the ANSI C compatibility mode's has specific advantages as indicated below:

Extended mode	CONVEX extensions to the language. Functions optimized for CONVEX hardware.
Conforming mode	POSIX functions.
Strict mode	Increased portability between ANSI C compilers.

These compatibility modes can be combined to tailor the compilation environment of an application.

Two command lines are required to compile and link an application in a mixed compatibility mode; both command lines use the `cc` command. The first command line translates the application into object code, specifying the language features and providing access to information in the appropriate header files. The second command line indicates which libraries are linked into the executable program.

Three language specifications can be used:

- Strict ANSI C
- ANSI C with CONVEX extensions
- Backward-compatible C

These language specifications are obtained by compiling an application with the strict, extended, and backward-compatible compatibility modes, respectively.

Similarly, there are four library systems:

- ANSI C functions
- ANSI C and POSIX functions
- ANSI C, POSIX, and CONVEX functions
- Backward-compatible CONVEX C functions

Linking with the strict, conforming, extended, and backward-compatible compatibility modes, respectively, provides access to these library systems. For example, you might want to develop a highly portable CONVEX program. Such a program uses functions that are specific to CONVEX hardware, but is strict in its interpretation of the language. Use the following command lines:

```
cc -D_POSIX_SOURCE -D_CONVEX_SOURCE -str -c applic.c
cc applic.o
```

The first command line provides function prototypes to CONVEX functions, while maintaining strict interpretation of the language. The _CONVEX_SOURCE macro can be used only if the _POSIX_SOURCE macro is defined, *even if no POSIX functions are used.* The second command line specifies that the ANSI C, POSIX, and CONVEX extension libraries are used.

Mode Conversion

This section lists the steps required to convert programs to the backward-compatible or the extended mode of the CONVEX C compiler.

Porting to Backward-Compatible Mode

Three steps are required to port applications compiled with the CONVEX C V3.0 compiler or the Common C compiler to the backward-compatible mode of CONVEX C:

1. Compile under ConvexOS
2. Use `lint` to remove errors
3. Recompile in backward-compatible mode

Step 1: Compile Under ConvexOS

You must compile the application under ConvexOS V8.1 or later. If the application was originally compiled using the Common C compiler, it is necessary to use the `pcc` command because CONVEX C V4.0 and later is invoked with the `cc` command line. Any problems found in this step are probably caused by incompatibility between ConvexOS and the environment in which the application was previously compiled and executed.

Step 2: Use `lint` to Remove Errors

The second step removes system-dependent code and code that can produce errors. This increases the portability of the application to CONVEX C. Refer to Chapter 4 for more information about the `lint` utility. The suggested command line for this step is:

```
lint -c -h *.c -pcc
```

where

`*.c`	represents names of source files.
`-c`	detects errors that occur in casting.
`-h`	applies heuristic tests to discover errors.
`-pcc`	runs `lint` in the backward-compatible mode.

Removing system-dependent code reduces chances for an error when CONVEX C is used, or when libraries that are linked with the program are modified.

Step 3: Recompile in Backward-Compatible Mode

The third step exposes errors caused by differences between the compiler used in the first step and the CONVEX C compiler. The command line for this step is:

```
cc -pcc *.c
```

where

`*.c`	represents names of source files.
`-pcc`	directs the compiler to execute in backward-compatible mode.

Some changes that can be required for programs that were compiled with the Common C compiler are listed in the section, "Incompatibilities of Common C," of this chapter.

Porting to Extended Mode

After an application has been ported to the backward-compatible mode of CONVEX C, it can be ported to the extended mode. There are three steps:

1. Link with extended libraries.
2. Use `lint` to remove errors.
3. Compile in extended mode.

Step 1: Link with Extended Libraries

Linking with ANSI C libraries removes the dependence on the old system libraries. Command lines for this step are:

```
cc -pcc -c *.c
cc *.o
```

where

`*.c`	represents names of source files.
`*.o`	represents names of object files.
`-c`	prevents the object files from being linked.
`-pcc`	directs the compiler to execute in backward-compatible mode.

The first command line compiles source files with the backward-compatible mode of the compiler. The second command line links resulting object files with functions available in the extended mode of the compiler.

Failures at this stage are probably caused by differences between standard library functions and backward-compatible library functions, or by the application's dependence on undocumented behavior of old library routines.

Incompatibilities between backward-compatible mode libraries and ANSI C libraries are listed in the "Extended Mode Differences" section of this chapter.

Step 2: Use `lint` to Remove Errors

The second step invokes the `lint` utility to remove system-dependent code and code that can produce errors. The suggested command line for this step is:

```
lint -c -h *.c -ext
```

where

`*.c`	represents names of source files.
`-c`	detects errors that occur in casting.
`-h`	applies heuristic tests to discover errors.
`-ext`	runs `lint` in the mode that accepts CONVEX extensions to the language.

Removing system-dependent code reduces chances for an error when CONVEX C is used, or when libraries that are linked with the program are changed.

Step 3: Compile in Extended Mode

The extended mode of the ANSI C compiler provides the ANSI C features, POSIX functions, and CONVEX extensions. The command line used at this step is:

```
cc *.c
```

where

`*.c` represents names of source files.

Failures at this step are probably caused by differences between ANSI C and the definition of C accepted by the backward-compatible mode of the compiler. These language differences are documented in the next section of this chapter.

Extended Mode Differences

The extended mode is the default compatibility mode of the compiler. This section discusses problems that can be encountered when porting programs from earlier versions of CONVEX C compiler or the Common C compiler. Problems include:

- Language features that prevent a non-ANSI C program from being compiled
- Changes in the semantics of the language
- Changes in header file organization
- Future directions in ANSI C
- Error return codes of C intrinsic functions

Most of these problems concern the translation of existing code to ANSI C. Information provided in this document can be used to assess difficulties in porting programs between systems. The incompatibilities between the Common C compiler and the backward-compatible mode of CONVEX C are discussed in "Incompatibilities of Common C," section of this chapter.

Changes that ANSI C imposes on existing applications can be grouped into two categories: changes that prevent compilation and changes that alter semantics of a construct without inhibiting compilation. Semantic changes are potentially more disruptive because different code can be produced without warning.

Chapter 9, "CONVEX C Intrinsics," addresses the problem of using C intrinsic functions.

Changes That Prevent Compilation

Changes introduced by the ANSI C standard are as follows:

- Five new keywords have been added: `const`, `enum`, `signed`, `void`, and `volatile`. Use of these words in the wrong context will generate an error message.
- Declaring an identifier that is common between two or more compilation units must obey the following rule: only one compilation unit can contain a definition of the identifier; the remaining compilation units must declare the identifier using the `extern` storage class.
- The numerals 8 and 9 are no longer available in octal constants.
- String literals cannot be modified.
- Converting a pointer of any object type (other than `void`) to a pointer of a different object type without an explicit type cast is not permitted.
- Function pointers cannot be converted to nonfunction pointers, and vice versa.
- Pointers to functions that have different parameter-type information are different types.
- `long float` is not in the ANSI C standard. Its use generates a warning in all ANSI C modes of CONVEX C.
- Functions called with a variable number of parameters must have a function prototype.
- The `entry` and `asm` statements are not accepted in ANSI C. The `asm` statement is available as a CONVEX extension. `entry` is not a CONVEX extension.
- Accessing a nonexistent member of a structure or union is an error.
- Empty declarations are invalid except for mutually referencing `struct` and `union` structures.
- Declaring zero-length arrays is invalid.
- No type specifiers can be added to a type that was defined using `typedef`.
- Formal parameters of a function cannot be `typedef` names.
- Predefined macro names, such as __FILE__ and __LINE__, cannot be redefined or undefined with the `#define` or `#undef` preprocessor directives.

Semantic Changes

A list of semantic changes introduced by the ANSI C standard follows. These semantic changes do not cause the compiler to generate error messages; they can cause a non-ANSI C program to generate incorrect output. Most of these changes are cited in the *American National Standard for Information Systems -- Programming Language C* document.

- A program that depends on preserving unsigned arithmetic conversions will behave differently, probably without complaint. This is considered the most serious semantic change that results from the current ANSI C standard.
- The compiler cannot reorder expressions that contain successive identical commutative or associative operators if the reordering can produce different results.
- Programs with character sequences that are trigraphs, such as `??!`, in string constants, character constants, or header names, are replaced by the corresponding character representation of such a trigraph. Trigraphs and characters they represent are listed in Table 3-2.

Table 3-2: Trigraph Representations

Trigraph Symbols	Represents
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

- A program relying on file scope rules for external declarations might be valid under block scope rules but behave differently when compiled in an ANSI C compatibility mode.
- Unsuffixed integer constants can have different types. Their type depends on the smallest integral type required to represent them.
- A constant of the form `'\078'` is valid, but denotes a character constant whose value is the combination of the values of `'\07'` and `'8'`.
- Because the escape sequences `'\a'` and `'\x'` have been added to ANSI C, a constant of the form `'\a'` or `'\x'` might have different meaning.
- It is neither required nor forbidden that identical string literals be represented by a single copy of the string in memory; a program depending upon either scheme might not behave the same as with a previous C compiler.

- To eliminate ambiguity, the following assignment operators no longer exist:

  ```
  =>>, =<<, =&, =^, =|, =+, =-, =*, =/, =%
  ```

  Further, for the assignment operators of the form +=, no space is permitted between the two characters + and =.
- Expressions with `float` operands can now be computed at lower precision. Compilers prior to the ANSI C standard performed all floating-point operations in `double`.
- Shifting by a `long` count no longer converts the shifted operand to `long`.
- Calculations in `#if` expressions might simulate either the translation environment or the execution environment. Consequently, a program that depends on properties of one particular environment might now give different answers.
- The empty declaration

  ```
  struct x;
  ```

  is no longer innocuous because it can now be used to hide a definition of `x` that exists in an outer block.
- Code that relies on a bottom-up parsing of aggregate initializers with partially elided braces do not yield the expected initialized object.
- Functions that depend on `char` or `short` parameter types being widened to `int`, or `float` to `double`, might behave differently.
- A macro that relies on formal parameter substitution within a string literal produces different results.
- Comments in ANSI C are replaced with a blank; in Common C they are removed. If your code relies on `/**/` to paste tokens together in Common C, you must now use ## instead. ## is the paste operator in the ANSI C preprocessor.
- Function-like macros can be recursive, but they are not recursively expanded.
- Results of floating-point operations might not be the same due to differences in casting and rounding.

Header File Changes

The following is a list of changes in organization of header files. Differences are noted when a function or macro that is present in the CONVEX C V3.0 compiler or Common C compiler is not located in the same header file in an ANSI C mode of CONVEX C.

This section does not list differences between the backward-compatible mode of CONVEX C and the CONVEX C V3.0 compiler. These differences are noted in the section, "Incompatibilities of Common C," of this chapter.

References are made to CONVEX extensions and POSIX functions. These extensions and functions are available in the appropriate compatibility modes of CONVEX C discussed in the "Description of Four Modes" section in this chapter.

Changes in organization of header files are:

ctype.h — The four function-like macros `isascii`, `toascii`, `_toupper`, and `_tolower` do not exist in the ANSI C standard. However, they do exist as CONVEX extensions.

math.h — Single-precision math functions (`sfabs`, `ssqrt`, `shypot`, `scabs`, `ssin`, `scos`, `stan`, `sasin`, `sacos`, `satan`, `satan2`, `sexp`, `slog`, `spow`, `ssinh`, `scosh`, `stanh`) are available only in the backward-compatible mode of the compiler. These math functions are available as function-like macros in the extended compatibility mode.

Macro definitions HUGE and HUGEI, while present in the backward-compatible mode, have been replaced by the macro definition DBL_MAX in the ANSI C modes.

signal.h — Macro names for the signals SIGCLD, SIG_CATCH, and SIG_HOLD are no longer available. The function-like macro SIGNALS_IN_PROG does not exist. These four macros are available only in the backward-compatible mode of CONVEX C.

stdio.h — Functions `fileno` and `fdopen` are POSIX functions.

strings.h — This header file no longer exists. All its functions are now contained in the string.h header file.

The functions `creat`, `close`, `lseek`, `open`, `read`, `write`, and `unlink` are available as POSIX extensions.

Signal handlers must execute signal(*signal*, SIG_DFL) before they process a signal.

Future Directions

Expected changes to the C language are listed below. While some of these changes might not occur, you should be aware of them to avoid possible incompatibilities in the future. These are taken from *American National Standard for Information Systems -- Programming Language C* document.

Changes concerning the C language are listed below:

- Restriction of the significance of an external name to fewer than 31 characters or to only one case is an obsolete feature that is a concession to existing C compilers.
- Lowercase letters as escape sequences in character and string literals are reserved for future standardization.
- Old-style function declarations that have empty parentheses will be obsolete.
- Two parameters declared with an array type (prior to their adjustment to pointer type) cannot refer to the same or overlapping objects. The `-alias array_args` option of the compiler enforces a stronger version of this

requirement. Its use reduces the necessity for the `no_recurrence` optimization directive. Refer to Chapter 2, "Compiler Fundamentals," for more information on the `-alias array_args` compiler option.

Changes to library specifications that can be expected in a future standardization of ANSI C are listed below. These changes are taken from the *American National Standard for Information Systems — Programming Language C* document.

errno.h
: Macros that begin with "E" and a digit or "E" and an uppercase letter (followed by any combination of digits, letters, and underscore) might be added to the declarations in the errno.h header.

ctype.h
: Function names that begin with either "is" or "to", and a lowercase letter (followed by any combination of digits, letters, and underscore) might be added to the declarations in the ctype.h header.

locale.h
: Macros that begin with "LC_" and an uppercase letter (followed by any combination of digits, letters, and underscore) might be added to definitions in the locale.h header.

math.h
: Names of all existing functions declared in the math.h header, suffixed with "f" or "l", might be used for corresponding functions with `float` and `long double` arguments and return values.

signal.h
: Macros that begin with either "SIG" and an uppercase letter or "SIG_" and an uppercase letter (followed by any combination of digits, letters, and underscore) might be added to definitions in the signal.h header.

stdio.h
: Lowercase letters might be added to the conversion specifiers in fprintf and fscanf. Other characters may be used in extensions.

stdlib.h
: Function names that begin with "str" and a lowercase letter (followed by any combination of digits, letters, and underscore) might be added to declarations in the stdlib.h header.

string.h
: Function names that begin with "str", "mem", or 'wcs" and a lowercase letter (followed by any combination of digits, letters, and underscore) might be added to declarations in the string.h header.

Incompatibilities of Common C

This section lists the incompatibilities between the backward-compatible mode of CONVEX C and the Common C compiler.

The incompatibilities are divided into two sections:

- Language definition
- Command line

Most of the incompatibilities between the two compilers result from illegal code that is accepted by the Common C compiler. CONVEX C does not accept illegal C code in any compatibility mode.

Language Definition

Type Specifiers

The Common C compiler permits type specifiers to modify a data type created with `typedef`. For example,

```
typedef int my_int;
typedef unsigned my_int u_my_int;
```

This is illegal C code; such specifiers are not allowed in any compatibility mode of CONVEX C.

Mixing static and extern

Mixing `static` and `extern` in a declaration is undefined in the C language; such declarations are not portable between compilers.

Multiple Initializers

The Common C compiler erroneously permits simple variables to have multiple initializers. For example,

```
{
    extern int init1(), init2(), init3();
    int x = { init1(), init2(), init3() };
}
```

When compiled with Common C, three functions are executed, with the last one assigning its return value to **x**. Multiple initializations are not permitted with the CONVEX C compiler in any of its compatibility modes.

Casts

The Common C compiler permits objects on the left side of an assignment operation to be cast. For example,

```
{
    int x;
    (int)x = 10;
}
```

This is allowed in Common C only when the cast is the same type as the variable or is a pointer to the same type as the variable. This is an error in the Common C compiler. CONVEX C detects this error.

Order of Evaluation

The order of expression evaluation used by the two compilers is not the same. This impacts numerical computations and may change the order in which side effects occur. The changes can cause a program that produced correct results with the Common C compiler to produce incorrect results with the CONVEX C compiler. The program is not a valid C program because it depends on the order of evaluation in ways not specified by the language. For example, given:

```
#include <stdio.h>

int a[2] = {0, 1};
int b[2] = {3, 4};

main(){
     int i = 0;
     a[i++] = b[i]; /* order of eval. undefined */
     printf("a[0] = %d\n", a[0] );
}
```

If compiled with `/bin/pcc file.c`, this program displays `a[0] = 3`, but if it is compiled with `cc -pcc file.c`, it displays `a[0] = 4`. The `lint` program can detect this type of error. Refer to Chapter 4 for a brief introduction to the `lint` program.

Uninitialized Variables

Applications that depend on uninitialized variables, dangling pointers, and other poor programming practices might not produce the same results on both compilers.

Negative Bit Shifts

The Common C compiler allows bit shifts to use negative operands. For example, a right shift with a negative right operand is equivalent to a left shift with a positive right operand. Negative operands of bit shift operators generate errors in the CONVEX C compiler if the shift length is a constant.

Switch Statements with Pointers

The Common C compiler allows the expression in a `switch` statement to have a pointer value. The CONVEX C compiler reports an error if the expression does not have integral type.

Undefined Functions

The Common C compiler silently converts functions to the `extern` storage class if they are declared `static` and used in a compilation unit but not defined. The CONVEX C compiler produces a warning when it performs this conversion.

Functions Returning short int or char

Undeclared functions that return a `short int` or `char` have their return value automatically widened in the calling routine, to an `int` by the Common C compiler; this matches the default declaration of a function. CONVEX C does not perform this automatic conversion. Programs that depend on this behavior produce erroneous results. However, `lint` is capable of finding these errors.

Command Line Differences

The `-t` compiler option provided by the Common C compiler is silently ignored by CONVEX C. The semantics of the `-B` compiler option are slightly different between the two compilers.

Chapter 4
Program Development Tools

This chapter describes some tools that assist in program development. These utilities, which make the development process easier, include:

- `lint` — checks for errors the compiler does not detect.
- `make` — makes program compilation easier and eliminates redundant compilations.
- `indent` — formats C source files for easier reading.
- `error` — inserts error messages into source files.

lint Utility

The `lint` utility is a program that checks C source files for code that can cause bugs or reduce the portability of a program. It also performs more rigorous type-checking than most C compilers. Types of errors that can be detected using `lint` include unreachable statements, loops not entered at the top, and automatic variables declared and not used. Consistency in the use of functions is also checked to find functions that return values in some places and not in others, and functions called with varying numbers of parameters. Similarly, each use of an object is compared with its declaration for consistency.

With the introduction of the CONVEX C V4.0 compiler, the disparity between the level of type checking performed by the compiler and `lint` is reduced. Although the ANSI C standard introduces more stringent type checking, it does not perform type checking across compilation units. In contrast, `lint` performs type checking across compilation units. For example, `lint` detects the error in the following code, but the C compiler does not:

```
/* file: one.c */
#include <stdio.h>
extern short int x;
int main()
{
    (void) printf("val = %d\n", x );
    return(0);
}

/* file: two.c */
long int x = 3;
```

Note that the above code is in two separate files. The problem is that in file one.c, `x` is declared to be a `short int` whereas in file two.c, `x` is declared to be a `long int`.

`lint` also detects unreachable ("dead") code. For example, the following code contains a statement that is never accessed when the program is executed:

```
#include <stdio.h>

int main()
{
        goto skip;
        (void) printf("this statement is not ever printed");
 skip: return(0);
}
```

If `lint` is used, the `printf` statement is recognized as unreachable. Dead code that `lint` discovers can be the result of a logical error in the program design.

Some of the options available with the `lint` utility are as follows:

- `-D -I -U` Same as for the `cc` compiler command. Refer to Section "Preprocessor Options" in Chapter 2 for a description of these options.
- `-a` Report assignments of long values to shorter variables.
- `-b` Report `break` statements that cannot be reached. (This is not the default because most `lex` and many `yacc` outputs produce dozens of such statements.)
- `-c` Complain about casts that have questionable portability.
- `-d` This option provides improved control over `lint`'s error messages. Refer to Appendix F, "Error Messages," for more information on how to use this option.
- `-ext` Run `lint` in a mode compatible with `cc`'s default compatibility mode.
- `-h` Apply heuristic tests to try to find bugs and improve style.
- `-n` Do not check compatibility to the standard library.
- `-pcc` Run `lint` in a mode compatible with `cc`'s backward-compatible mode.
- `-std` Run `lint` in a mode compatible with `cc`'s conforming mode.
- `-str` Run `lint` in a mode compatible with `cc`'s strict mode.
- `-u` Ignore functions and variables used and undefined, or defined and unused. This is suitable for running `lint` on a subset of files out of a larger program.
- `-v` Suppress complaints about unused arguments in functions.
- `-x` Report variables that external declarations refer to, but never use.
- `-z` Do not complain about structures that are never defined.

`lint` also includes some directives that change the output of the `lint` utility:

- /*NOTREACHED*/ At appropriate points, suppresses comments about unreachable code.
- /*VARARGS*n**/ Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/	Turns on the `-v` option for the next function.
/*LINTLIBRARY*/	Used at the beginning of a file; shuts off complaints about unused functions in the file.

For information on this utility, refer to the `lint(1)` man page and the *CONVEX Guide to Software Development.*

make Utility

The `make` utility uses the time and date stamps of source and object files to decide whether a file must be compiled. It compiles only those programs that have been modified since the last version of the program was compiled. Thus, if a program depends on twelve separate compilation units, but only two of them have been modified, only those two are recompiled before the entire program is loaded.

Consider the following code for a `make` file:

```
#                                                    1
# Make file for a short example.                     2
#                                                    3
myprog: myprog.o second.o                   #        4
        cc myprog.o second.o -o myprog      #        5
                                            #        6
myprog.o: myprog.c local.h                  #        7
        cc -c myprog.c                      #        8
                                            #        9
second.o: second.c local.h pivot.h          #        10
        cc -c second.c                      #        11
```

If this file is stored in a file named makefile, the executable program `myprog` can be created by entering the command **make**. The first three lines contain comments; comment lines start with a "#" symbol. Line 7 states that myprog.o is dependent on the two files myprog.c and local.h. If these two files are modified after myprog.o was created, the command on the line 8 is executed. This line creates an up-to-date version of myprog.o. The remaining lines follow the same format. Consequently, if pivot.h is changed, commands on line 11 and line 5 are executed to create an executable file, `myprog`.

This example, while useful enough for small programs, uses only a few of the features that the `make` utility provides. Information on the additional features can be found in the *CONVEX Guide to Software Development* and in the `make(1)` man page. The `-k` option of the C compiler generates file dependencies that can be used in a makefile.

indent Utility

The **indent** utility formats C source files. Numerous options specify various printing styles, such as placement of comments and location of curly braces after **if** statements. This utility is useful because it converts different styles into one, consistent style, which can make programs easier to read.

The current version of **indent** does not recognize ANSI C keywords.

For example, consider the following code contained in the file indentin.c:

```
#include <stdio.h>
int main(){int j,i;j=5;i=6;printf("j=%d,i=%d",j,i);return(i*j);}
```

This small program is unreadable; it has no white space to help identify the hierarchy of the program. The following command line helps to untangle this program.

```
indent indentin.c indentout.c -bad -bc -di8
```

where

`-bad`	inserts a blank line after every block of declarations.
`-bc`	inserts a newline character after each comma in a declaration.
`-di`*n*	specifies the indentation, in *n* character positions, from a declaration keyword to its following identifier.

The contents of the indentout.c file are as follows:

```
#include <stdio.h>
int
main()
{
        int     j,
                i;

        j = 5;
        i = 6;
        printf("j=%d,i=%d", j, i);
        return (i * j);
}
```

Thus, poorly indented code can be converted to a readable form.

The **indent** utility provides many more options that you can use to format a program. For example, the **-troff** option formats the input file for the **troff**(1) utility. Consult the **indent**(1) man page for more information on the **indent** utility.

error Utility

The `error` utility analyzes and optionally inserts the diagnostic error messages compilers and language processors produce, such as `make` and `lint`, into the source file and line where the errors occurred. It replaces the painful, traditional methods of scribbling abbreviations of errors on paper, and permits you to view error messages and source code simultaneously without manipulating multiple windows in a screen editor.

`error` looks at the error messages, either from the specified file name or from the standard input, and attempts to determine the following:

- Language processor that produced each error message
- Source file and line number to which the error message refers
- Whether to ignore the error message

After all input has been read, `error` then inserts the error message, which might be slightly modified, into the source file as a comment on the line preceding the line to which the error message refers.

Error messages that cannot be categorized by language processor or content are sent to the standard output. You can use several options with this utility:

`-q`	Confirm any potentially dangerous action (such as modifying a file) or verbose action.
`-t` *suffixlist*	Do not touch files whose suffixes appear in the suffix list. The suffix list is separated by dots, and "*" wildcards are accepted.
`-v`	After inserting error messages into all the files, initiate the `vi` editor to edit the first file containing an error.
`-T`	Produce terse output.

The following `csh` command line checks the syntax of a C source file, inserts any error messages into the file, and places the file in the `vi` editor if errors are detected.

```
cc -sc file.c |& error -v
```

The equivalent command for the Bourne shell (`sh`) is as follows:

```
cc -sc file.c 2| error -v
```

Chapter 5
Debugging Programs

Introduction

This chapter introduces utility programs you can use to find errors in a program. Some of the programs discussed are optional products. If you are unsure whether a program exists on your system, ask your system manager. The utilities presented include the following:

- `CXdb` — a window oriented symbolic debugger
- `csd` — a symbolic debugger
- `pmd` — postmortem dump
- `cref` — a cross-reference generator
- `adb` — an assembly-language debugger

An overview of each of these utilities is provided in this chapter; details appear in references provided in respective sections. The purpose of this chapter is to provide a description of the tools that are available, not to present an in-depth tutorial.

CXdb Debugger

CONVEX CXdb, the visual debugger, is an optional debugger that has all the debugging functions found in traditional debuggers. To generate the information necessary for using CXdb, you must compile your program with the `-cxdb` option on the command line.

Like the debugger provided with the CONVEX Consultant, CXdb can perform these functions:

- Debug program source code or disassembled code
- Debug programs containing multiple source modules
- Access program variables by name

In addition to these capabilities, CXdb can perform these functions:

- Provide debugging contexts for source code and disassembled code in a windowing environment
- Attach CXdb to a running process
- Execute debugger commands while your process is running
- Create aliases and macros to simplify debugger commands
- Debug optimized code

CXdb's windowing environment supports line-oriented terminals and workstations capable of displaying CX/Motif. This windowing environment eases the task of debugging programs that contain multiple threads of execution. Refer to *CXdb Concepts* for additional information on CONVEX CXdb.

CONVEX Consultant Debuggers

CONVEX Consultant is an optional product that includes a package of utilities you can use to debug and analyze the performance of C or FORTRAN programs. Some utilities that CONVEX Consultant includes are as follows:

- Symbolic debugger
- Postmortem dump

The *CONVEX Consultant User's Guide* describes each program and tells you how to use it effectively. The guide also discusses proper techniques to follow when debugging optimized or parallelized code.

Symbolic Debugger

The CONVEX symbolic debugger (`csd`) is a tool designed specifically for debugging C and FORTRAN programs running under the ConvexOS operating system. To generate the information necessary for using `csd`, you must compile your program with the `-db` option on the compiler command line.

The `csd` program lets you perform the following tasks:

- Debug a program at the source level or at the assembly-language level
- Examine core dumps and obtain a symbolic runtime stack trace
- Debug programs containing multiple source modules
- Access program variables by name rather than by absolute address
- Debug optimized code

To support parallel processing, `csd` provides special debugging commands to assist in monitoring program execution on multiple processors. You can use these commands to determine how many threads exist, control the number of threads permitted, determine the current thread, change the current thread, and examine the communication registers by which threads communicate.

Postmortem Dump

The postmortem dump (`pmd`) generates information to assist your debugging efforts if the program running under it aborts and dumps memory. To run a program under `pmd`, you must first compile the program using the `-db` option on the compiler command line.

At your option, `pmd` produces either a short-form dump or a long-form dump containing the information shown in Table 5-1. The short-form dump is the default.

Table 5-1: Postmortem Dump Contents

Type of Dump	Contents
Short Form	Signal that caused the program to abort Runtime stack backtrace Approximate source line location at which the exception occurred
Long Form	Signal that caused the program to abort Runtime stack backtrace Approximate source line location at which the exception occurred Contents of the machine registers Dump of active local variables in each routine on the runtime stack Dump of global, or common, variables Region of disassembled object code where the exception took place Summary of resources used by the program

The summary of resources produced by the long-form dump includes execution time, elapsed time, percent of time in CPU, size of shared memory and unshared memory, page faults, and swaps.

For more information about this utility, refer to the *CONVEX Consultant User's Guide*.

Cross-Reference Generator

The cross-reference generator (`cref`) produces a table of references to each object in a C source program. These objects include all user-defined C structures and variables. The format of the table is controlled by options specified on the `cref` command line.

Consider the following C source program, crefex.c.

```
 1 #define loop_incr 200
 2 #define array_size 1000
 3
 4 extern void sub1(int);
 5 float a[array_size];
 6
 7 struct some_struct {
 8   int a;
 9   int b;
10 };
11
12 struct some_struct var_struct = {
13   10,
14   20
15 };
16
17 void main()
18 {
19   int i;
20
21   for( i=1; i<=array_size; i+=loop_incr )
22     sub1(a[i]);
23   var_struct.a = loop_incr;
24   return(0);
25 }
```

The cross-reference generator produces a listing in the following format when

cref crefex.c

is executed:

```
% cref crefex.c

a             5#   8#   22   23
array_size    2#   5    21
b             9#
i            19#  21    22
loop_incr     1#  21    23
main         17#
some_struct   7#  12#
sub1          4#  22
var_struct   12#  23

Symbols = 9      Hash table size = 20      Density = 0.450000
```

The first column of the output contains the name of the object. The numbers following each object state where that object is used. Pound signs (#) indicate lines on which an object is modified.

The `cref` output in the previous example illustrates some limitations of the `cref` utility that you should be aware of. First, the object `some_struct` is shown by cref to be defined twice, on line 7 and on line 12. Actually, the object is defined on line 7, but creates a variable on line 12. However, the `cref` utility interprets both uses of object `some_struct` to be a definition of that object. Second, consider the `a` object. `cref` indicates that this object is also defined twice. However, the source code contains two `a` objects. One is an array; the other is a member of a structure. Consequently, to avoid confusion, do not use the same name for several objects.

`cref` is useful as long as its limitations are recognized. If you execute `cref` on a program that is syntactically incorrect, results are unpredictable. For a complete description of the cross-reference generator, refer to the `cref(1)` man page.

Assembly-Language Debugger

The assembly-language debugger, `adb`, is an object-code debugger that requires no recompilation or special compiler options. The CONVEX `adb` debugger allows you to examine core dumps from failed programs and to interactively debug programs at the assembly-language level.

Because programs are run under `adb`, it is always aware of the state of the program and values of all variables. Using `adb`, you can perform the following functions:

- Display the assembly-language instructions of the program
- Stop program execution at any point
- Examine values of program variables
- Modify the value of any program variable
- Execute a program one instruction at a time
- Display values of machine registers
- Modify values of machine registers

You can use the `adb` debugger can be used to debug programs at all optimization levels, including vector code and programs running on multiple processors. For a detailed description of `adb` and complete instructions about its use, refer to the *CONVEX adb Debugger User's Guide*.

Chapter 6
Profiling Programs

Introduction

This chapter discusses several tools you can use to increase the performance of a program. Profilers analyze programs to detect code that uses the greatest time. Programs do not usually contain the information required by the profiler, so the compiler needs to add additional code, called instrumentation, to the executable program when it is being compiled. Profilers presented in the remainder of this chapter use the instrumentation in different ways.

CONVEX Consultant Profilers

The CONVEX Consultant package is an optional product that includes a package of routines you can use to debug and analyze the performance of C or FORTRAN programs. This package offers three profilers that allow you to monitor the performance of your program:

- Standard profiler (`prof`)
- Basic block profiler (`bprof`)
- Graph profiler (`gprof`)

You can use the information obtained from a profiler to improve the efficiency and speed of a program. To use a specific profiler, you must first compile your program using one of the options described in Table 6-1.

Table 6-1: Compiler Options for Profiling

Compiler Option	Description
-p	Produces code that counts the number of times each routine is called. When the program begins execution, the monitor utility is called. If the program completes normally, a profile file (mon.out) is produced. This file can be processed by the prof profiler to generate an execution profile. When you specify the -p option, the linker searches profiling libraries instead of the standard libraries.
-pb	Produces code that counts the number of times each statement is executed. If the program completes normally, a basic block profile file (bmon.out) is produced. This file can be processed by the bprof profiler to display the source-level execution counts.
-pg	Produces instrumentation similar to the -p option, and invokes a runtime recording mechanism that keeps more extensive statistics. If the program completes normally, a call graph profile file (gmon.out) is created. This file can then be processed by the gprof profiler to produce a comprehensive execution profile.

For further information about these optional profilers, refer to the *CONVEX Consultant User's Guide.*

CXpa Profiler

The CONVEX C compiler supports the CXpa profiler (Version 2.0 or later) on CONVEX C200 series computers. This is an optional product.

CXpa provides a more accurate profiler than `prof` and `bprof`. This performance analyzer is described in detail in the *CONVEX CXpa User's Guide.*

CXpa is an interactive tool that can monitor program activity at the routine level, the loop level, or the block level.

- Routine-level profiling produces summary information about routines that are called during profiled execution of the program. This information includes:

  — Number of times the routine is called
  — Wall-clock time spent in the routine and percentage of program total wall-clock time
  — CPU time spent in the routine and percentage of program total CPU time
  — Net CPU time spent in the routine and percentage of program net CPU time

- Loop-level profiling produces summary information about individual loops in the program. Certain loop optimizations affect the way a loop is profiled. For example, if loop distribution, partial vectorization, or dynamic selection has been performed, each replicated copy of the loop is profiled separately. Information provided for a loop includes:

  — Type of loop (scalar, vectorized, or parallelized)
  — Number of times the loop is executed and the vector length
  — Total CPU time in the loop

- Block-level profiling shows how many times each basic block in your code is executed. A basic block is a set of sequential assembly-language statements, the last of which changes the flow of control.

Chapter 7
Optimizing C Programs

This chapter gives an overview of the optimization, vectorization, and parallelization features that are built into CONVEX C. These features improve the efficiency of your program and increase its execution speed. You can control the extent to which the compiler optimizes your program through command line options and pragmas discussed in this chapter.

There are two versions of the CONVEX C compiler: a scalar optimizing version bundled with each CONVEX machine and a vectorizing/parallelizing version, which is an optional product. The scalar compiler cannot perform vector and parallel optimizations, and it ignores compiler options associated with vectorization and parallelization.

For a more complete discussion of optimization concepts and techniques, refer to the *CONVEX C Optimization Guide.*

Introduction

Optimization transforms functional code into an executable program that returns correct answers in less time than code that has not been optimized. The CONVEX C compiler automatically performs the following transformations:

- Eliminate unnecessary operations
- Perform operations in a more efficient order
- Replace certain operations with faster ones

These transformations operate at several different optimization levels. Each level adds a new degree of optimization to optimizations performed at lower levels. Optimization can be grouped into two categories: machine-dependent and machine-independent. The machine-independent optimizations are scalar optimizations, vectorization, and parallelization.

Scalar optimizations are performed at the basic-block (local) or function (global) level. Basic-block scalar optimizations involve transformations of code within a sequence of consecutive source code statements that have only one entrance and one exit. Function-level scalar optimizations involve transformations of code within a single program unit (that is, main program or function). The `-O0` command line option causes the compiler to perform basic-block scalar and machine-dependent optimization. The `-O1` command line option causes the compiler to perform function-level and basic-block scalar optimizations as well as machine-dependent optimizations.

Vector optimization greatly improves performance of programs that manipulate arrays. For example, in a loop that adds the corresponding elements of two arrays, vector registers can be used to perform single-instruction additions on up to 128 elements at a time. Specifying the `-O2` option on the `cc` command line causes the compiler to perform vector and scalar optimizations.

Unlike vector optimization, which reduces CPU usage, parallel optimization improves program throughput by allowing multiple CPUs to participate in processing. Invoked with the `-O3` option, the compiler generates code that allows your program to be executed by as many CPUs as are available when the program runs. Like automatic vector optimization, automatic parallel optimization works on loop constructs in your program. Parallel optimization can increase performance roughly proportional to the number of CPUs on your system. Under the `-O3` option, parallel, vector, and scalar optimizations are performed.

Scalar Optimization

The CONVEX C compiler automatically performs many optimizations on scalar code. Scalar optimizations fall into the following categories: machine-dependent optimizations, machine-independent basic-block optimizations, and machine-independent function-level optimizations.

Machine-dependent optimizations are performed at optimization level `-no` and higher. At optimization level `-O0`, the compiler also optimizes code across a basic block, which is a group of statements delimited by a single entrance and a single exit. At optimization level `-O1`, the compiler optimizes code across a program unit.

The following subsections describe transformations the compiler performs *automatically* when you compile a program for scalar optimization.

Machine-Dependent Optimization

Machine-dependent optimization generates object code that takes advantage of the CONVEX machine architecture. Most of the machine-dependent optimizations in the following list are performed on all optimization levels:

- **Instruction scheduling**—The compiler reorders instructions to use the functional units on the computer most effectively. Instruction scheduling on a CONVEX machine reorders instructions within statements at all optimization levels, across multiple statements at level `-O0` and higher, and across a group of basic blocks at optimization level `-O1` and higher.
- **Span-dependent instructions**—The compiler tries to generate a 2-byte branch or a 4-byte jump instruction for conditional and unconditional transfers of control within a program. These short-form instructions conserve memory and improve execution speed.
- **Register allocation**—The compiler tries to maximize the number of registers allocated for a given expression.

- **Hoisting**—At optimization level -O1, the compiler "hoists" a scalar or array reference out of an innermost loop if the value does not change within the loop. At optimization level -O2, an array reference can be hoisted out of a vectorized loop if the array is indexed only by loop constants and the loop control variable.
- **Strength reduction**—The compiler performs certain strength-reduction operations on instruction-level operations. For example, the compiler can transform integer multiplication into addition: X*2 is replaced by X+X.
- **Tree-height reduction**—Internally, the compiler represents expressions as trees, the height of which corresponds to the depth of the expression. In general, the time required to evaluate an expression is proportional to the height of the tree. Unless prevented from doing so, the compiler chooses the evaluation order that yields the least depth.

Basic-Block Optimization

Basic-block optimization is machine-independent scalar optimization performed on a sequence of consecutive statements with one entrance and one exit. The -O0 option on the cc command line causes the compiler to perform basic-block optimization as well as machine-dependent optimization. Basic-block optimizations the compiler performs are as follows:

- **Redundant-assignment elimination**—This optimization removes unnecessary assignments to a given variable.
- **Assignment substitution**—The compiler substitutes the assigned value of a variable for subsequent uses of the variable, thus eliminating redundant loads.
- **Common subexpression elimination**—To avoid repetitious calculations, the compiler recognizes a common subexpression and retains its value in a register.
- **Redundant-use elimination**—The compiler detects multiple uses of a variable between two assignments to it and eliminates load instructions by retaining the value of the variable in a register.
- **Constant propagation and folding**—Constant propagation and folding is a form of assignment substitution. After assigning a constant to a variable, the compiler replaces subsequent uses of the variable with the constant.
- **Algebraic simplification**—The compiler simplifies certain algebraic and trigonometric expressions.
- **Simple strength reduction**—The compiler replaces time-consuming operations with those that execute faster.

Function-Level Optimization

Function-level optimization is machine-independent scalar optimization performed over a group of basic blocks, in particular, loops and conditional statements. The `-O1` option on the `cc` command line causes the compiler to perform function-level optimization as well as basic-block and machine-dependent optimization.

Function-level optimizations the compiler performs are as follows:

- **Constant propagation and folding**—Function-level constant propagation and folding is similar to basic-block constant propagation and folding, with one exception. If the variable is actually used later in the same group of basic blocks, the folded constant is propagated throughout that group.
- **Dead code elimination**—If, during constant propagation and folding, the arithmetic or logical expression of an `if` statement is folded to 0, the unreachable branch is eliminated. Also, code that is to be conditionally compiled is eliminated if the test variable is set to 0.
- **Copy propagation**—Copy propagation occurs when the compiler replaces a variable with another variable to which it has been equated. For example, in the statement `X = Y`, the compiler can replace later occurrences of `X` with `Y`.
- **Redundant assignment elimination**—Assignment statements that are not used elsewhere within a given group of basic blocks are removed.
- **Common subexpression elimination**—Function-level common subexpression elimination removes common subexpressions across a group of basic blocks. The value of the common subexpression is retained in a register or temporary variable. Subsequent occurrences are replaced by references to the register or temporary variable.
- **Code motion**—Invariant computations in a loop are moved above the loop.
- **Strength reduction**—Certain operations on loop induction variables and loop constants are replaced by operations that execute faster.

Vector Optimization

The CONVEX C compiler automatically optimizes a program to use vector hardware on CONVEX machines. Vector optimization, commonly called vectorization, converts scalar operations on arrays into equivalent vector operations. Vector operations use the vector registers to perform identical operations on up to 128 array elements simultaneously.

The `-O2` option on the `cc` command line causes the compiler to perform vector optimizations as well as scalar optimizations.

The compiler reorders the statements and instructions of a program to make it easier to vectorize.

Explanations of the most important of these transformations are as follows:

- **Strip mining**—The vector registers hold up to 128 elements. If the number of iterations of a vectorizable loop exceeds 128, the compiler replaces the loop with two loops. The innermost loop, which has an iteration count that never exceeds 128, is vectorized.
- **Scalar expansion**—When a loop includes the use of scalar values in vector calculations, the scalar value is either stored in a register that can be fed repeatedly into the functional unit or the scalar value is propagated into elements of a vector register.
- **Vectorization of conditional loops**—Vectorization of loops containing `if` and `if-else` structures is achieved by executing all operations necessary to produce a vector of test results, performing calculations in all blocks of conditional code, and merging the results.
- **Loop distribution**—Nested loops can be vectorized by distributing the outermost loop and vectorizing each of the resulting loops or loop nests.
- **Loop interchange**—The compiler interchanges loops when necessary to ensure that the rightmost index is associated with the innermost loop, to achieve a more efficient vector length, or to remove a recurrence that prevents vectorizing an innermost loop.
- **Conditional induction variables**—A variable that changes linearly within the loop but is not incremented on every iteration. The compiler can frequently recognize such variables and generate vector code for expressions involving them.
- **Reductions**—The compiler vectorizes a special class of recurrences known as reductions.

The following types of loops cannot be vectorized:

- Loops that contain function calls
- Loops with an iteration variable whose start value, stop value, or step value varies within the loop
- Loops that have multiple entries or exits
- Loops that are preceded by the `scalar` or `no_vector` pragmas
- Loops that contain `goto` statements
- Loops that contain real recurrences

Parallel Optimization

The CONVEX C compiler parallelizes and vectorizes to enhance program performance. Parallelization differs from vectorization in that it does not reduce CPU use. Instead, it spreads processing of a single program across multiple CPUs, improving wall-clock time for that program.

The `-O3` option on the `cc` command line causes the compiler to perform parallel optimizations in addition to vector and scalar optimizations.

The compiler automatically parallelizes the outermost loop in a nest (which can be the strip-mine loop created when a loop has been vectorized), if it can be safely parallelized. The compiler also distributes and interchanges loops to generate parallel code for the outer loop. Most scalar reductions and assignments can be parallelized at the cost of some additional synchronization code.

Parallelization and vectorization are closely related. As with vectorization, the presence of any of the following can inhibit or prevent parallelization:

- Multiple entries or exits
- Function calls
- Loop-carried dependencies

The compiler does not automatically parallelize a loop containing a function call. You can force it to do so by preceding the loop with the `force_parallel` pragma. This pragma forces the compiler to parallelize the immediately following loop, regardless of potential dependencies the compiler detects. Certain actual dependencies (such as from one scalar to another) cause the compiler to ignore the pragma. Also, functions called from within a parallelized loop should be compiled using the `-re` option. Each invocation of a function compiled with the `-re` option maintains a thread-private stack to store compiler-generated temporary variables and generates multiple copies of loops that are conditionally executed, depending on whether the program runs in parallel. For more information on this and other optimization pragmas, refer to the "User-Directed Optimization" section of this chapter.

A loop-carried dependency (LCD) exists when an assignment in one iteration stores a value that is used during a subsequent or previous iteration. Since such a computation is inherently serial, it cannot be parallelized. If the number of iterations is sufficiently large, however, you can improve loop turnaround time by directing the compiler to allow multiple CPUs to participate in its serial processing. To do so, precede the loop with the `synch_parallel` pragma. This pragma causes the compiler to generate code that divides the loop iterations among available CPUs and correctly handles potential errors at the boundaries of those divisions. For synchronized code to execute efficiently in parallel, the portion of the loop without dependencies must be large relative to the portion of the loop that contains dependencies. For more information on the `synch_parallel` pragma and other optimization pragmas, refer to the "User-Directed Optimization" section of this chapter. For more information on loop-carried dependencies, refer to the *CONVEX C Optimization Guide.*

The Optimization Report

When you compile your program with the `-O2` or `-O3` command line options, the compiler generates an optimization report for each program unit. This report consists of a loop table or an array table, or both.

For example, consider the following matrix multiplication loop:

```
 1 int main(){
 2  int n = 200;
 3  float a[200][200], b[200][200], c[200][200];
 4  int i,j,k;
 5
 6  for( i=0; i<n; i++ )
 7   for( j=0; j<n; j++ ) {
 8     c[j][i] = 0.0;
 9     for( k=0; k<n; k++ )
10        c[j][i] = c[j][i] + a[k][i] * b[j][k];
11    }
12   return(0);
13 }
```

Notice that the individual array elements `c[j][i]` are summed directly (at line 10) rather than stored in a temporary scalar variable. Introduction of a temporary scalar—later assigned to `c[j][i]`—would inhibit vectorization.

The following screen shows the optimization report output generated by compiling the file example1.a at optimization level `-O3`. No array table is generated for this program.

```
% cc -O3 -or loop -tm c2 main.c

     Optimization by Loop for Routine main

Line   Iter.  Reordering          Optimizing / Special    Exec.
Num.   Var.   Transformation       Transformation          Mode
--------------------------------------------------------------------------
  6    i    Dist
  6-1  i    FULL VECTOR Inter
  6-2  i    FULL VECTOR Inter
  7-1  j    PARALLEL
  7-2  j    PARALLEL
  9-2  k    Scalar

Line   Iter.  Analysis
Num.   Var.
--------------------------------------------------------------------------
  6-1  i    Interchanged to innermost
  6-2  i    Interchanged to innermost
```

Line numbers in the `Line Num.` column are number pairs, indicating that at least one loop was distributed. In this example, loop distribution creates two loop nests, called distributed parts. In each pair, the first number is the source file line number; the second number indicates the orphan.

The loop table lists the optimizations performed on each loop and consists of two parts. The first part of the loop table shows that these transformations were performed:

- The `i` loop at line 6 was distributed; both distributed parts were interchanged and fully vectorized.
- The `j` loop in both distributed parts was parallelized.
- The `k` loop in the second distributed part was not transformed.

The second part of the loop table provides additional information about the transformations performed. The compiler reports that the `i` loop in each distributed part was interchanged to innermost, so that it could be vectorized.

User-Directed Optimization

You can control the extent to which the compiler optimizes your program by using command line options and pragmas discussed in this section.

Command Line Options

CONVEX C provides several command line options that allow you to select the level of optimization to be performed on your program. These options apply to the entire source file being compiled unless overridden by one of the optimization pragmas. Section "Optimization Options" in Chapter 2 describes the optimization options. If you do not specify one of the optimization options on the command line, the compiler defaults to the `-no` level.

Optimization Pragmas

A pragma provides information that the compiler cannot determine or instructs the compiler to override conditions that automatically control optimization, vectorization, or parallelization. A compiler pragma has the form

#pragma _CNX *pragma-name* [*options*]

Table 7-2 lists the pragmas. Most pragmas affect the loop that immediately follows them. The compiler ignores pragmas that it does not recognize. Refer to Appendix B, "Pragmas," for a description of each pragma.

Table 7-1: Optimization Pragmas

Pragma	Description
`begin_tasks`	Identifies the beginning of a group of independent tasks for parallel execution.
`end_tasks`	Terminates a group of independent tasks for parallel execution.
`force_parallel`	Forces a loop to be parallelized.
`force_parallel_ext`	Forces a loop to be parallelized, possibly interchanging outer loops for vectorization.
`force_vector`	Forces a loop to be vectorized.
`max_trips` (*n*)	Defines the maximum number of trips that will be made through a loop.
`next_task`	Identifies each individual task in a group of independent task for parallel execution.
`no_parallel`	Suppresses parallelization of a loop.
`no_recurrence`	Disregards apparent recurrences.
`no_side_effects`(*func*[, *func*])	Specifies that one or more functions have no side effects.
`no_vector`	Suppresses vectorization of a loop.
`prefer_parallel`	Identifies to the compiler the loop you prefer to have parallelized. This is useful with nested loops.
`prefer_parallel_ext`	Identifies to the compiler the loop you prefer to have parallelized. This pragma can interchange outer loops for vectorization.
`prefer_vector`	Identifies to the compiler the loop you prefer to have vectorized. This is useful with nested loops.
`pstrip` (*n*)	Defines the parallel strip-mine length for a loop.
`scalar`	Prevents a loop from being vectorized or parallelized.
`select`(-,-,-)	Generates multiple versions of a loop and selects at runtime the version to execute based on specified trip counts.
`synch_parallel`	Generates parallel code for a loop, inserting synchronization code to honor dependencies.
`unroll`	Reduces loop overhead by causing replication of the loop body.
`vstrip` (*n*)	Defines the vector strip-mine length for a loop.

Chapter 8
Mixed-Language Programming

This appendix explains how to access routines written in other languages. Conventions used to call routines written in C are defined. Accessing routines in other languages based on the C model of function-calling is discussed.

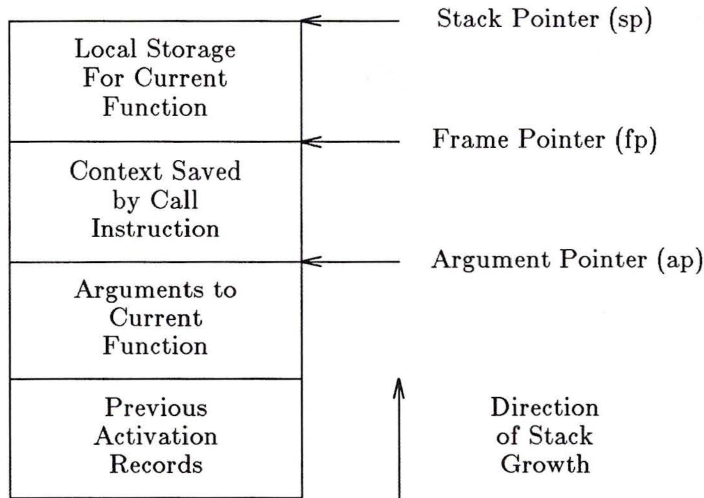
C Function Calls

When C function calls are executed, the current state of three hardware registers used by the calling function must be preserved. These registers are the processor status word, the frame pointer (fp), and the argument pointer. Contents of these registers are pushed on the runtime stack as a part of an activation record. The only register that the called function *must* preserve is the frame pointer register.

Function Stack Layout

Figure 8-1 shows the top of the runtime stack. The stack pointer (sp) register contains the address of the topmost location on the runtime stack. The fp register contains the address of the last frame pushed on the runtime stack by a `call` or `calls` assembly-language instruction. The argument pointer (ap) register contains the address of the arguments to the current function.

Figure 8–1: Top of the Runtime Stack



Standard Calling Sequence

A function call requires the following steps:

1. Push values of arguments to the function on the runtime stack in reverse order.

2. Update the ap register. The updated register points to the first argument in the argument list. The first argument in the list is the last one pushed.

3. Push an additional word. This word contains the number of arguments passed.

4. Call the function with a `calls` assembly-language instruction.

A `calls` instruction places a stack frame on the runtime stack. The stack frame contains current values of the program counter (pc), the processor status word (psw), the frame pointer (fp), and the argument pointer (ap). The fp is set equal to the sp and the sp is updated to point to the new top of stack.

These conventions apply to function calls:

- The called function can allocate storage for local variables on top of the runtime stack. No stack references in CONVEX C code are made relative to the top of the runtime stack. Storage a called function allocates on the stack is automatically deallocated when the function returns.

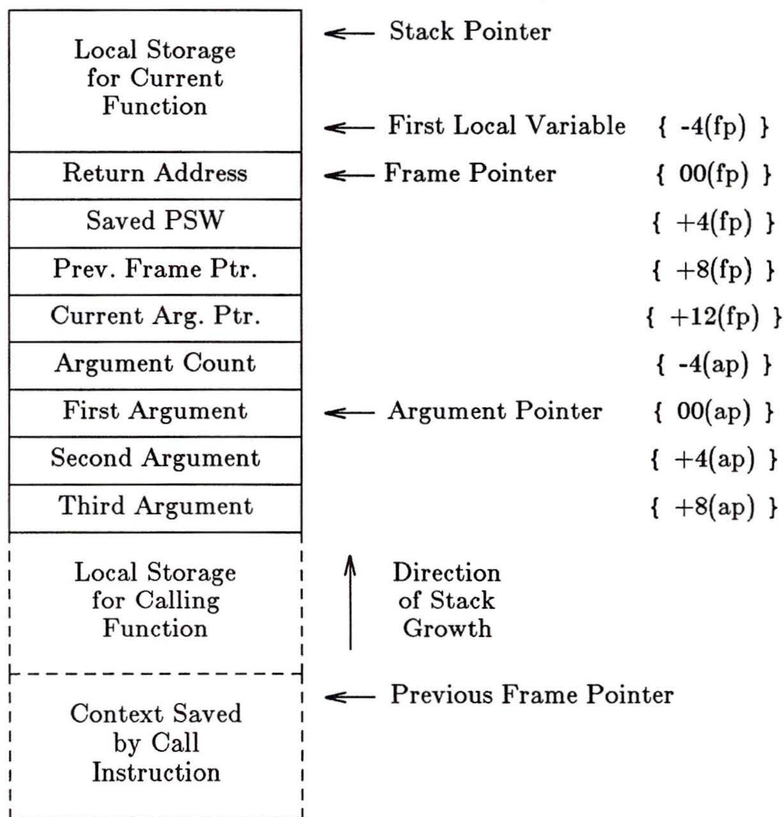
- The called function need not preserve the contents of any register except the fp. The called function uses the current value of the ap to access arguments its parent passes to the function.

- The fp points to the context block that contains the return address, the saved machine registers, and the function arguments that the caller pushes on the stack. The called function references the local storage it has allocated on the runtime stack by negative offsets from the fp.

- The called function references arguments as positive offsets from the ap. The word with an address of -4(ap) contains the number of arguments passed to the function.

Figure 8-2 shows the layout of the stack as seen by a function after it has been called and after it has allocated some storage for local variables on top of the runtime stack. The stack is shown as a series of 32-bit words.

Figure 8–2: Stack Layout



Called functions return by placing a return value in register s0, then executing the `rtn` instruction. When the `rtn` instruction is executed, automatic storage the called function allocates is automatically deallocated. This instruction restores the processor status word register and the frame pointer to their previous states, then returns control to the location immediately following the `calls` instruction that called the function.

After control returns, the stack pointer register points to the location that contains the pushed argument count. The parent function adds a positive number offset to the value in the stack pointer to remove the argument count and any pushed arguments. The value added is the total number of bytes pushed before the call. Finally, before the parent can access any of its own arguments, it reloads its own argument pointer register from the current frame on the stack. This value is at 12(fp).

Assembly Code Generated for Standard Calls

The following example shows a section of CONVEX assembly-language code used for a function call. Generally, C functions are called as shown in the example.

Example:

```
psh.w  lastarg          ; value of rightmost argument
psh.w  otherargs        ; value of arguments in
psh.w  otherargs        ; reverse order
psh.w  firstarg         ; value of first argument
mov    sp,ap            ; arg pointer points at first arg
pshea  #argcount        ; push count of # of args passed
calls  _child           ; use 'calls' to call function
add.w  #bytecount,sp    ; remove bytes pushed for args
ld.w   12(fp),ap        ; reload our argument pointer
```

The code shown below is used in the called child function:

```
    .globl _child           ; called function
_child:
    sub.w  #lo              ; assuming 32-bit size
    ld.w   (-4)ap,s1        ; load count of the args passed
    st.w   s0,-4(fp)        ; first local int is at -4(fp)
    sub.w  s0,s0            ; value returned in s0
    rtn                     ; return to caller
```

Standard Function Names

Function names and global variables produced by the compiler in object code are limited to 32 characters. An underscore character is prefixed to each global variable. Include this character when using the assembly-language debugger or when writing assembly language functions to be called by C functions.

Standard Function Arguments and Return Values

CONVEX C passes arguments to functions by value rather than by reference; that is, the value of the argument, rather than its address is passed. However, arrays are passed by reference. To pass arguments by reference, use either pointer data types or the & address operator.

When structures are passed by value, all elements of the structure are pushed onto the runtime stack before the call.

Scalar register s0 contains results returned from functions. ANSI C permits structures to be returned from functions. Because the memory required by a structure may be larger than the s0 register, when a structure is returned from a function, the structure is stored in the data area and its address is returned in scalar register s0.

Accessing FORTRAN Routines from C

This section provides information required to call FORTRAN functions from C:

- Translation of FORTRAN function names to C function names
- C data types equivalent to FORTRAN data types
- FORTRAN parameter passing protocol
- FORTRAN protocol for function return values

Function Names

If you are calling a FORTRAN routine, you must follow FORTRAN naming conventions. FORTRAN variables, arrays, and functions have symbolic names that must begin with a letter and can be followed by letters (`A-Z`), digits (`0-9`), underscores (_), or dollar signs ($) up to a maximum length of 40 characters. C source code should not use uppercase letters because FORTRAN identifiers with external linkage use lowercase letters only.

The FORTRAN compiler appends an underscore to FORTRAN subroutine and function names. Because the C compiler does not append this character, use the full function or subroutine name in C. For example, you would access a subroutine in FORTRAN named `ADDEM` from C using the name **`addem_`**.

Data Representations

Table 8-1 shows the corresponding FORTRAN and C declarations that can be used as function parameters. In FORTRAN, all variables declared as INTEGER, LOGICAL, or REAL (without *) default to the same amount of memory, 32 bits.

This memory size can be changed with a compiler option.

Table 8-1: FORTRAN and C Declarations

FORTRAN	C
	typedef unsigned long long int UINT;
LOGICAL*1 x	signed char x;
LOGICAL*2 x	signed short int x;
LOGICAL*4 x	signed long int x;
LOGICAL*8 x	signed long long int x;
INTEGER*1 x	signed char x;
INTEGER*2 x	signed short int x;
INTEGER*4 x	signed long int x;
INTEGER*8 x	signed long long int x;
REAL*4 x	float x;
REAL*8 x	double x;
REAL*16 x	struct { UINT sign:1; UINT exp:15; UINT umant:48; UINT lmant; } x;
COMPLEX*8 x	struct {float real, imaginary;} x;
COMPLEX*16 x	struct {double r, i;} x;
CHARACTER*6 x	signed char x[6];

The logical true value in FORTRAN (`.TRUE.`) consists of 1's in each bit position, while the logical false value (`.FALSE.`) consists of 0's in each bit position.

Both languages use two's complement to represent integers. Both languages also use either CONVEX native or IEEE format to represent floating-point numbers. To use IEEE format, your machine must have IEEE support hardware.

When passing floating-point data to FORTRAN subroutines, use the same floating-point format (native or IEEE) in the calling routine and the called routine. VECLIB, a collection of optimized FORTRAN-callable numerical subprograms, can automatically detect the floating-point format being used. (Refer to the *VECLIB User's Guide* for more information on this optional software package.)

Parameter Passing

Parameters are passed to FORTRAN routines by reference only. This means that the contents of a variable are accessible in a FORTRAN routine only if the address of the variable is passed to the routine. This is simple for the primitive data types: prefix the & operator in front of the variable name. Pointers require the pointer name; arrays are automatically passed by reference.

Because FORTRAN has no data type equivalent to the C `struct` data type, the only time that a structure should be passed to a FORTRAN routine is when the receiving FORTRAN variable is the COMPLEX*8 or COMPLEX*16 data type. To avoid alignment difficulties, do not use other structure data types.

For example, consider the following synopsis for a FORTRAN routine:

```
INTEGER add1, add2, sum
ADDEM( add1, add2, sum )
```

This subroutine adds two numbers and returns the result in the variable named `sum`. The correct method to call this subroutine from C is:

```
int add1, add2, sum;
addem_( &add1, &add2, &sum );
```

Note the underscore following the FORTRAN function name. The FORTRAN compiler automatically appends this character to all FORTRAN functions. The **&** operator is prefixed to the front of the variable name so that the contents of the variable are accessible to the FORTRAN routine.

Because CHARACTER data types are equivalent to C `char` arrays, they are passed by including the name of the array in the parameter list. However, the length of the character array must also be appended as a long integer to the actual parameter list:

FORTRAN:

```
CHARACTER*4 name
CHARACTER*10 other
PROC_NAME( name, other )
```

C:

```
char string[4];
char other[10];
proc_name_( string, other, 4L, 10L );
```

Use caution in passing arrays with multiple dimensions to FORTRAN because that language stores arrays in column-major order, while C stores arrays in row-major order.

To pass complex data types to FORTRAN, you must use a C `struct` data type shown in Table 8-1. An example that passes complex numbers follows:

FORTRAN:

```
COMPLEX*8 add1, add2, sum
ADDEM( add1, add2, sum )
```

C:

```
struct { float r, i; } add1, add2, sum;
addem_( &add1, &add2, &sum );
```

You can to pass a `struct` data type because the alignment of the real and imaginary parts in FORTRAN memory coincide with two `float` data types in C.

FORTRAN Routines

FORTRAN functions are similar to C functions. FORTRAN subroutines are similar to C functions that return the `void` data type. FORTRAN functions that do not return COMPLEX*8, COMPLEX*16, or CHARACTER data types are declared in the normal fashion, but these three data types require that you add a parameter to the parameter list of a function.

For example, the function

```
INTEGER add1, add2
INTEGER ADDEM( add1, add2 )
```

requires the following function declaration for ANSI C:

```
extern int addem_( int *add1, int *add2 );
```

and the following function declaration for C programs compiled in the backward-compatible mode of CONVEX C:

```
extern int addem_( );
```

The C keyword `extern` indicates that the object code for `addem_` is located in another compilation unit. The parameter list for this function is empty because the backward-compatible mode of the C compiler does not recognize function prototypes.

FORTRAN functions return complex data types (`COMPLEX` and `CHARACTER`) in an implied first argument before the list of parameters declared by the FORTRAN function. Make sure the first argument is a pointer to a structure that contains the function result when the call returns.

For example, if a FORTRAN function returns a `COMPLEX*16` data type

```
COMPLEX*16 add1, add2
COMPLEX*16 ADDEM( add1, add2 )
```

C requires a void function declaration:

```
struct d_complex { double dr, di; };
extern void addem_( struct d_complex *sum, struct d_complex *add1,
    struct d_complex *add2 );
```

The second argument in the C function declaration corresponds to the first argument in the FORTRAN function declaration.

Even though the original FORTRAN function returns a value, the C function is void because the value the FORTRAN function returns is accessed as a parameter in the C function declaration. The function declaration for a program that is compiled using the backward-compatible mode of CONVEX C is as follows:

```
extern void addem_( );
```

FORTRAN functions returning a CHARACTER data type called from a C source file require a C function declaration that returns a void type with two additional parameters in the parameter list. The format of the parameter list in this case is:

function_name(string address, string length, other parameters)

For example, the FORTRAN source code

```
CHARACTER*15 GET_ERROR( )
```

requires the following ANSI C function declaration:

```
extern void get_error_( char error[], long int charlen );
```

The following C source code is used to call this FORTRAN function:

```
char error_type[15];
get_error_( error_type, 15L );
```

Note that L is appended to the number in the function call because that number *must* be a `long int`.

FORTRAN Input and Output

Do not call FORTRAN routines that perform I/O from an application that is initialized as a C program. The methods that the two languages use to perform I/O are incompatible; mixing the I/O of both languages causes undefined behavior.

Accessing Ada Routines from C

Ada routines cannot be called from a C application. CONVEX C does not support access to CONVEX Ada routines. However, CONVEX Ada supports access to CONVEX C functions. For information on calling C routines from an Ada application, refer to the *CONVEX Ada User's Guide.*

Chapter 9
CONVEX C Intrinsics

This chapter defines intrinsic functions and intrinsic instructions and identifies behavior they can introduce.

What Are Intrinsics?

In CONVEX C, there are two types of intrinsics: intrinsic instructions and intrinsic functions. Intrinsic instructions are commands in a CONVEX instruction set. For example, an instruction in a CONVEX C100 Series architecture is `add`; an instruction in a CONVEX C200 Series architecture is `sqrt`. While the `add` instruction is in the instruction set for a C200, the `sqrt` instruction is not implemented in the C100.

Intrinsic functions are useful because they:

- Require less function overhead
- Execute faster than non-intrinsic functions
- Do not inhibit vectorization of loops

The disadvantages of intrinsic functions are:

- Math intrinsic functions do not modify `errno` when an error occurs.
- The compiler does not recognize a dependency between `errno` and the math intrinsic functions.

Intrinsic functions are C callable functions that may use intrinsic instructions. Intrinsic functions are used by default in the extended compatibility mode of the compiler. You can obtain access to these functions in the strict and standard compatibility modes by including `-U__NO_INLINE_MATH` on the `cc` command line. If you want to use intrinsic functions in the backward-compatible mode, include `-D__INLINE_MATH` on the `cc` command line or include the fastmath.h include file in the source code.

The following example is a C program that calls `sqrt`, an ANSI C function.

```
#include <math.h>
#include <stdio.h>
int main()
{
    int x = 4;
    int y;
    y = sqrt(x);
    (void) printf("%d\n", y );
    return(0);
}
```

If this program is compiled and executed on a C1 machine or compiled with the `-tm C1` command line option, an intrinsic `sqrt` function is used with the program. This is because the C1 instruction set does not have a `sqrt` instruction; the `sqrt` function must be implemented using a software square-root algorithm. But if it is compiled and executed on a C2 machine, or compiled with the `-tm C2` or `-tm c2s` command line options, the intrinsic `sqrt` instruction is used.

You can see which functions are implemented as intrinsics by searching the include files for the `__NO_INLINE` macro. Intrinsic functions are implemented as function-like macros defined with CONVEX reserved function names.

For example, the math.h include file contains the following code fragments:

```
#   if !defined(__NO_INLINE) && !defined(__NO_INLINE_MATH)
     /* fast implementations of the routines defined by ANSI C */
#     define acos(x) _mth$d_acos((double)(x))
#     define asin(x) _mth$d_asin((double)(x))
      .
      .
      .
#   endif
```

In this example, the `acos` function is a function-like macro that calls `_mth$d_acos`.

> **Note** Do not call functions that define intrinsic functions directly in your programs. These function names are subject to change.

Intrinsic Function Behavior

Generation of Signals

Intrinsic functions may not modify the `errno` variable when an error occurs. When an intrinsic instruction detects an error it does not generate a signal because when a C program begins execution, the bit in the program status word register that controls the generation of intrinsic error signals is set to 0.

For example, when the following program is compiled in the extended compatibility mode and executed on a C2 machine, an incorrect result is printed:

```
#include <math.h>
#include <stdio.h>
#include <errno.h>
int main()
{
    int x = -4;
    int y;

    errno = 0;
    y = sqrt(x);
    if( errno == EDOM )
        (void) printf("domain error\n");
    else
        (void) printf("%d\n", y );
    return(0);
}
```

When this program is compiled for a C1 machine using the `-tm c1` command line option and executed, the domain error is caught because the implementation of the `sqrt` intrinsic function on a C1 modifies the `errno` variable when an error occurs. On a C2, however, the `sqrt` intrinsic function *does not* modify `errno`.

errno and Optimization

Another problem associated with the use of intrinsic functions is that at levels of optimization higher than `-no`, the compiler removes redundant code or replaces code with faster pieces of code. This can cause a problem with some intrinsic functions. For example, the compiler removes the conditional statement in the following code because it does not realize that the `acos` function can modify `errno`:

```
errno = 0;
a = acos(x);
if(errno == EDOM) {
        ...
}
```

At optimization level `-O2`, this is optimized to the following:

```
a = acos(x);
```

How to Disable Intrinsics

Only the math intrinsic functions are not accessible by default in the strict and standard compatibility modes. The reason for this is that ANSI C compliant programs expect `errno` to be modified when certain function errors occur. Similarly, the math intrinsic functions are not accessible by default in the backward-compatible compatibility mode to maintain compatibility with previous compilers.

There are two ways to prevent all intrinsic functions from being used by your program. The first way is to define the `__NO_INLINE` macro on your command line using the `-D` command line option. This prevents all intrinsic functions from being accessed. However this might be too stringent. There are several different types of intrinsic functions, and it may be necessary to disable only one type.

The eight types of intrinsic functions are listed below:

- `__NO_INLINE_BINT`
- `__NO_INLINE_CTYPE`
- `__NO_INLINE_MATH`
- `__NO_INLINE_SIGNAL`
- `__NO_INLINE_STDIO`
- `__NO_INLINE_STDLIB`
- `__NO_INLINE_STRING`
- `__NO_INLINE_TIME`

Each of these macros is named after the include file in which it can be found.

For example, the stdio.h include file, which contains some I/O functions, declares function-like macros defined with intrinsic functions. You can disable these intrinsic functions by including the `-D__NO_INLINE_STDIO` option on the command line. You might want to link your program with the intrinsic functions when you have completely debugged it.

The second way to avoid the **errno** problem is to set the intrinsic error enable bit of the program status word when your program is started. You must include a signal handler that determines what caused the signal and then take appropriate actions. Further detail of this approach is beyond the scope of this chapter. Refer to the *CONVEX Architecture Reference*, Chapter 3, "Register Sets," for more information on the program status word and the bits associated with intrinsic instruction errors.

Chapter 10
Input and Output

This chapter describes methods that a program uses to receive input and generate output. First, basic concepts of input and output functions are explained, then an example is presented. Functions used for input and output are defined briefly; refer to the man page for each function for additional information.

File Input and Output Concepts

This section of the chapter defines concepts used in file input and output (I/O). These concepts are generalized later to include program I/O.

File Manipulation

Accessing a file consists of three phases:

- Opening access to the file
- Performing input and/or output on the file
- Closing access to the file

When a file is opened, a data structure of type FILE is created to transfer data to or from a file. Many routines use this data structure to transfer data between files and a program. After the file is processed, you must close the file so that all buffers containing data for that file are flushed. If you do not perform this final step, you can lose data.

You can open a file can be opened with the `fopen` function. This function returns a pointer to a FILE data structure.

Example:

```
#include <stdio.h>

main()
{
    FILE *fp;
    fp = fopen("some_file", "r" );
             .
             .
             .
}
```

This statement opens the file some_file to read data. The include file, stdio.h, contains standard input and output function prototypes. The FILE data type declares a file pointer that is used by other I/O routines.

After opening the file, you can write text to it usin the **fprintf** function (and many other functions).

Example:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    FILE *fp = fopen( "output.data", "w" );
    if( fp == NULL ){
        (void) fprintf( stderr, "Can't open file: output.data\n" );
        exit( EXIT_FAILURE );
    }
    (void) fprintf( fp, "This is some sample text.\n" );
    (void) fclose( fp );
}
```

The data structure, pointed to by **fp**, contains status information about the file, such as whether an error occurred when the file was last accessed. All information contained in the data structure is accessible by library functions. One such function is `ferror`. For example,

Example:

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    FILE *fp = fopen( "output.data", "w" );
    if( fp == NULL ){
        (void) fprintf( stderr, "Can't open file: output.data\n" );
        exit( EXIT_FAILURE );
    }
    (void) fprintf( fp, "This is some sample text.\n");
    if( ferror( fp ) )
        (void) fprintf( stderr, "Error when writing to"
            " output file.\n" );
    (void) fclose( fp );
    exit( EXIT_FAILURE );
}
```

The `ferror` function checks for an error when the file associated with structure **fp** was accessed. The error condition exists until it is cleared with the `clearerr` function. There are many other functions you can use to manipulate files.

Although FILE is defined in the stdio.h header file, its contents must be accessed only with standard functions, not as a `struct`.

Finally, the `fclose` function closes access to and flushes the buffers of a file.

File Types and Access Modes

ANSI C defines two file types: binary and text. On a CONVEX computer system and all POSIX compliant systems, these two file types are identical. You do not need to be concerned with these two types unless you intend to port the program to another computer system. If that is the case, use the binary or text types as required by the other computer system.

The access mode for a file is defined when access to a file is opened. The three basic modes of operation for files are reading, writing, and appending. Reading examines the contents of a file, without modifying them. Writing modifies the contents of a file. Appending data to a file adds data at the end of a file without changing the original data. Other modes are created by combining these basic modes. For example, the read/write mode permits data in a file to be read as well as written. The method used to obtain each of the access modes is defined explicitly in the fopen(3) man page.

A program can access files only if the program has access permission for the file. For example, if the file only has read-only permission, an error occurs if the program tries to open that file for writing, appending, or reading and writing. For more information on file permissions refer to the chmod(1) man page.

System Functions and Stream Functions

Two groups of functions are provided with CONVEX C: system I/O functions and stream I/O functions. The stream I/O functions use the system I/O functions to perform more complicated tasks.

System I/O functions provide simple, basic file I/O functions. The are:

- `close`
- `creat`
- `lseek`
- `open`
- `read`
- `write`

In contrast, there are many more stream I/O functions. The stream I/O functions are described in Section "stdio.h" in Chapter 11. Some of these functions are used to manipulate the size of a file buffer to enhance performance, while others are used to obtain specific pieces of data in a file. For example, the `fscanf` function can extract a number from the middle of a line of text.

Thus, the group of I/O functions used in a program depends on the sophistication required to manipulate data files. Also, system I/O functions limit portability of a program because they are not available in the ANSI C standard.

Program Input and Output

You can perform program I/O on files as well as devices. Every C program has three files that are automatically available: `stdin`, `stdout`, and `stderr`. These files are used for standard input (keyboard), standard output (display), and standard error output (display), respectively. Some device names can be found in the /dev directory; the devices that begin with the "tty" prefix are terminals.

Access the three files, `stdin`, `stdout`, and `stderr`, using stream functions only. For example, to print a sentence on a display, use the following command:

```
#include <stdio.h>

(void) fprintf( stdout, "This sentence will be printed"
        " out on the display.\n");
```

You can access many other devices from a program. Names of these devices are contained in the directory /dev on ConvexOS. Refer to the *CONVEX UNIX Primer* for more information on these devices.

ConvexOS supports redirection of input and output between files and C programs.

Example:

```
% myprog < myprog.in
```

This command forces all input for the executable file `myprog` to come from `myprog.in`. In this case, the device `stdin` is associated with the file `myprog.in` instead of the keyboard. Similar associations occur when you redirect files by piping, which permits the output of one program to be used as the input of another program. Refer to *ConvexOS Manpages for Users* for more information on this technique.

A useful stream I/O function is `freopen`. This function reassigns I/O from a device to a file.

Example:

```
#include <stdio.h>

(void) freopen( "stderr.file", "w", stderr );
```

This program fragment forces all succeeding output to the device `stderr` to be stored in the file named `stderr.file`. This function is useful in reassociating `stderr` with a particular error file.

Program Input and Output Example

The example below shows the use of stream I/O functions. It includes a short description of the program followed by the program source code. A line-by-line description of the program follows the example. Line numbers in the program are provided for reference only; they are not part of the source code.

This program shows one way to transfer data structures between a file and a program. It reads some information from a file into an array of structures, then appends it to the file in reverse order. Key routines that make the I/O of data structures possible are `fread` and `fwrite`.

```
1       #include <stdlib.h>
2       #include <stdio.h>
3       #define NUM_PHONES 5
4       int main()
5       {
6               FILE *fp;
7               int i,j;
8               struct {
9                       char name[8];
10                      char phone[7];
11              }
12              phone_array[NUM_PHONES];
13
14              if( (fp = fopen( "phones", "a+")) == NULL) {
15                (void) fprintf(stderr,"Unable to open file: phones\n ");
16                exit( EXIT_FAILURE );
17              }
18              if( fseek( fp, 0L, SEEK_SET ) != 0 ) {
19                (void) fprintf( stderr, "error seeking file origin\n" );
20                exit( EXIT_FAILURE );
21              }
22              j = fread( (void *)phone_array, sizeof( phone_array ),
23                              1, fp );
24
25              if( j != 1 ) {
26                (void) fprintf( stderr, "Error reading phone array\n" );
27                exit( EXIT_FAILURE );
28              }
29
30              if( fseek( fp, 0L, SEEK_END ) != 0) {
31                (void) fprintf( stderr, "error seeking end of file\n" );
32                exit( EXIT_FAILURE );
33              }
34
35              for( j=0, i=NUM_PHONES-1; i>=0; i-- )
36                j += fwrite((void *)&phone_array[i],
37                               sizeof(phone_array[i]), 1, fp );
38
39              if( j != NUM_PHONES ) {
40                (void) fprintf( stderr, "Error writing phone array\n");
41                exit( EXIT_FAILURE );
42              }
43              (void) fclose( fp );
44              return(0);
45      }
```

1 The first line includes necessary functions and data structures for input and output. The stdlib.h header file includes the function prototype for the exit function.

2 The stdio.h header file includes the definition of the `FILE` structure.

3 Declare macro constant `NUM_PHONES`.

4 Start the definition of the main function.

6 `fp` is a data structure that retains information on a file.

7 Declare two integers `i` and `j`.

8-11 These lines define the data structure that associates a phone number with a name.

12 `phone_array` is the array of data structures to be used for input and output in this program.

14 This line tries to open the file that contains names and phone numbers. The access mode is `a+` because the file is initially read, then data is appended to the file. This access mode does not permit destruction of any data. If the file cannot be opened, perhaps due to incorrect access permissions, `NULL` is returned. If `NULL` is detected, the program halts.

18 The `fseek` function in this line positions the file position pointer to the beginning of the file. This is required because the append mode positions this pointer to the end of the file when access to the file is opened. This function returns a zero if it is successful; otherwise, a nonzero return value indicates an error occurred. Consequently, the program verifies that no error occurred.

22-23 The function on these two lines reads in the five sets of phone numbers. `fread` requires four parameters: the address of a structure that receives the data, the number of bytes for each structure, the number of times the structure is read, and the stream pointer. The fread function returns 1 if reading was successful.

25 This conditional ensures that `fread` reads the required data.

30 The `fseek` function in this line returns the file position pointer to the end of the file. Again, the return value is verified to ensure that no error occurred.

35-37 A `for` loop is required to write the data structures because they are appended to the contents of the file in reverse order. Consequently, data structures must be written one at a time. Every time the `fwrite` function is called, the first parameter contains the address of each data structure in the array.

39 Like the `fread` function, the `fwrite` function returns the number of elements sent to the designated file. This line verifies that all structures elements are written and prints an error otherwise.

43 This line flushes the buffer associated with the `fp` data structure, and closes the connection with the file.

`fread` and `fwrite` transfer blocks of data of known size. All they require is a pointer to the data structure and its size. Thus, I/O is fast. But, there are disadvantages to this approach. For example, the data structure must be of a constant size. In the previous program, the name field was limited to 8 characters. Strings of variable length cause problems; they must be read using other more explicit methods that do not use a storage area with a fixed size. One such function is `fgets`.

Data files created by the functions `fread` and `fwrite` are not portable to other computer systems. One cause of this nonportability is the structure padding referred to in Appendix A, "Data Types and Representations." Structure padding occurs when the compiler aligns data types on boundaries in memory. This can cause blank spaces to appear inside a data structure. These blank spaces accompany the data structure when it is written out using `fwrite`. For example, computer A may insert one blank byte between the `name` array and the `phone` array in the previous program, while computer B may insert no spaces. When computer B attempts to read computer A's output, data is lost.

Consequently, do not use these functions when data files are to be transferred to other computer systems.

Chapter 11
Runtime Library

This chapter provides a brief overview of functions that are available in the C libraries provided with CONVEX C; ANSI C, POSIX, and CONVEX functions are included. The information is organized by header files.

Under each header file, the functions are listed with a short description and some common macro definitions.

The header files discussed in this chapter are:

- assert.h
- ctype.h
- errno.h
- float.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stdarg.h
- stddef.h
- stdio.h
- stdlib.h
- string.h
- time.h

Each section describes the ANSI C functions, associated structures, identifiers, and macros. Where applicable, the CONVEX and POSIX extensions that are accessible with these ANSI C header files are also described. The compatibility mode of the compiler determines which extensions are available.

CONVEX C provides four compatibility modes as shown in Table 11-1.

Table 11-1: Compatibility Modes

Mode	Language	Default Functions
Extended	ANSI C, CONVEX	ANSI C, CONVEX, POSIX
Conforming	ANSI C	ANSI C, POSIX
Strict	ANSI C	ANSI C
Backward-compatible	Common C	CONVEX

POSIX refers to a group of standards sponsored by various working committees of the IEEE. The Portable Operating System Interface for Computer Environments IEEE Std 1003.1-1988 (POSIX.1) is the first of the POSIX standards to be adopted. It was ratified on August 22, 1988, and represents a standard system call interface and environment based on the UNIX operating system. It is intended to support application portability at the source-code level.

By avoiding non-standard features of an operating system, an application has a greater chance of being ported to another computer system without major modifications. Refer to Chapter 3 for more information on the compatibility modes.

Functions Versus Function-like Macros

Function-like macros are very similar to functions; a function-like macro is called in the same manner as a function. There are several differences:

- Function-like macros may increase the size of the code.
- Function-like macros are often faster; there is no function call overhead such as register saving.
- Function calls may reduce optimization.
- Contents of functions have local scope.
- Functions have an address.
- Functions evaluation their arguments only once.

Function-like macros are most suitable for routines that are used often but require little code; such routines include character input and output functions. Several of the header files, ctype.h and stdio.h in particular, include both functions and function-like macros for some of the routines, permitting the programmer to choose which version should be used. When a library function can be accessed by both functions and function-like macros, all function calls access the function-like macro by default.

Two ways to access the function instead of the macro are:

- Undefine the function-like macro using **#undef**.
- Surround the function name with parentheses.

For example, in the header file ctype.h, **isalpha** is declared as a function prototype and defined as a function-like macro. Two ways of calling the function are

```
#include <ctype.h>

#undef isalpha
int ch = isalpha('A');
```

and

```
#include <ctype.h>

int ch = (isalpha)('A');
```

The macro is accessed as usual:

```
#include <ctype.h>

int ch = isalpha('A');
```

Thus, it is possible to select either a function or a function-like macro when a choice exists.

Functions that have function-like macros are identified in each function description.

Calling Runtime Functions

There are two ways to access a library function:

- Include the associated header file.
- Implicitly declare the function if no types from the header file are required.

Each method has its advantages and disadvantages.

Including a header file is easy and portable. Any data types, structures, and macros that are required to use the function are automatically declared and defined. When an application is ported to another system, changes to data structures in header files probably do not require changes to the source code. this is the preferred method.

For example, use the `memmove` function to replace the last four elements of `arra` with the first four elements.

```
#include <string.h>

char arra[10];

memmove( arra, &arra[6], 4 );
```

In the absence of a function prototype, a function is implicitly declared when it is encountered. Integral promotions are performed on each integral argument (that is, `char` and `short` are promoted to `int`), and arguments that have type `float` are promoted to type `double`.

In conclusion, header files are not always required to access a runtime function, but their inclusion is easy and assists in maintaining the program.

assert.h

This header file contains a function-like macro that halts a program if its argument is not true. This macro is useful in the development of an application, to ensure that certain conditions exist.

ANSI C

The function-like macro is:

void assert(expression)
Aborts program if `expression` is false and `NDEBUG` is not defined. Writes file name and line number of the file with the error.

CONVEX Extension

void _assert(expression)
Equivalent to the assert function.

ctype.h

This header file contains character handling functions that are used for classifying character-coded integer values by table lookup. Each function returns a nonzero value for true or zero for false.

ANSI C

Functions declared are:

int isalnum(int ch)
Returns true for any letter or digit.

int isalpha(int ch)
Returns true for any letter.

int iscntrl(int ch)
Returns true for any control character. Control characters are nonprinting characters.

int isdigit(int ch)
Returns true for any decimal-digit character.

int isgraph(int ch)
Returns true for any printing character except space (' ').

int islower(int ch)
Returns true for any lower case letter.

int isprint(int ch)
Returns true for any printing character including space (' ').

int ispunct(int ch)
Returns true for any punctuation character. These characters are defined as all printing characters except spaces, digits, or letters.

int isspace(int ch)
Returns true for any space, tab, carriage return, newline, vertical tab, or form feed.

int isupper(int ch)
Returns true for any upper case letter.

int isxdigit(int ch)
Returns true for any hexadecimal digit.

int tolower(int ch)
Returns the corresponding lower case letter when the argument is an upper case letter. If the argument is not an upper case letter, it returns the argument unchanged.

int toupper(int ch)
Returns the corresponding upper case letter when the argument is a lower case letter. If the argument is not a lower case letter, it returns the argument unchanged.

Functions also declared as function-like macros are:

- isalnum
- isalpha
- isalpha
- isdigit
- isgraph
- islower
- isprint
- ispunct
- isspace
- isupper
- isxdigit

CONVEX Extensions

Function-like macros are:

int isascii(int ch)
Returns true for any ASCII character.

int toascii(int ch)
Returns the argument with all bits that are not part of the standard ASCII character turned off.

int _tolower(int ch)
Returns the same data as tolower, except it has a restricted domain and runs faster. If the argument to _tolower is not an upper case letter, its result is undefined.

int _toupper(int ch)
Returns the same data as toupper, except it has a restricted domain and runs faster. If the argument to _toupper is not a lower case letter, its result is undefined.

errno.h

This header file contains the definition of many macro constants for error conditions.

ANSI C

Two macros defined are:

EDOM
Contains the value that indicates a domain error.

ERANGE
Contains the value that indicates a range error.

An external identifier declared is:

errno
Contains a value indicating the most recent error. This identifier is accessed globally by several library routines.

float.h

This header file defines macro constants that indicate characteristics of floating-point data types.

ANSI C

Macros defined are:

DBL_DIG
Number of decimal digits in a number of type `double` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

DBL_EPSILON
Smallest number z of type `double` such that $1.0 + z \neq 1.0$.

DBL_MANT_DIG
Number of digits in the floating-point mantissa of the `double` type. The base is the value of `FLT_RADIX`.

DBL_MAX
Largest representable number of type `double`.

DBL_MAX_10_EXP
Largest integer z such that 10^z can be represented by a normalized number of type `double`.

DBL_MAX_EXP
Largest integer z such that $FLT_RADIX^{(z-1)}$ is a normalized floating-point number of type `double`.

DBL_MIN
Smallest normalized positive number of type `double`.

DBL_MIN_10_EXP
Smallest representable integer z such that 10^z can be represented by a normalized number of type `double`.

DBL_MIN_EXP
Smallest integer z such that $FLT_RADIX^{(z-1)}$ is a normalized floating-point number of type `double`.

FLT_DIG
Number of decimal digits in a number of type `float` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

FLT_EPSILON
Smallest number z of type `float` such that $1.0 + z \neq 1.0$.

FLT_MANT_DIG
Number of digits in the floating-point mantissa of the `float` type. The base is the value of FLT_RADIX.

FLT_MAX
Largest representable number of type `float`.

FLT_MAX_10_EXP
Largest integer z such that 10^z can be represented by a normalized number of type `float`.

FLT_MAX_EXP
Largest integer z such that $FLT_RADIX^{(z-1)}$ is a normalized floating-point number of type `float`.

FLT_MIN
Smallest normalized positive number of type `float`.

FLT_MIN_10_EXP
Smallest representable integer z such that 10^z can be represented by a normalized number of type `float`.

FLT_MIN_EXP
Smallest integer z such that $FLT_RADIX^{(z-1)}$ is a normalized floating-point number of type `float`.

FLT_RADIX
Base number system for exponent representation.

FLT_ROUNDS
Rounding mode for floating-point addition. CONVEX computers round to the nearest representable value.

LDBL_DIG
Number of decimal digits in a number of type `long double` that are not changed when that number is rounded to a floating-point number that can be represented on the computer.

LDBL_EPSILON
Smallest number z of type `long double` such that $1.0 + z \neq 1.0$.

LDBL_MANT_DIG
Number of digits in the floating-point mantissa of the `long double` type. The base is the value of FLT_RADIX.

LDBL_MAX
Largest representable number of type `long double`.

LDBL_MAX_10_EXP
Largest integer z such that 10^z can be represented by a normalized number of type `long double`.

LDBL_MAX_EXP
Largest integer z such that $\texttt{FLT_RADIX}^{(z-1)}$ is a normalized floating-point number of type `long double`.

LDBL_MIN
Smallest normalized positive number of type `long double`.

LDBL_MIN_10_EXP
Smallest representable integer z such that 10^z can be represented by a normalized number of type `long double`.

LDBL_MIN_EXP
Smallest integer z such that $\texttt{FLT_RADIX}^{(z-1)}$ is a normalized floating-point number of type `long double`.

limits.h

This header file defines macro constants that indicate sizes of integral data types.

ANSI C

Macros defined are:

CHAR_BIT
Number of bits in `char` data type.

CHAR_MAX
Maximum value for an object of type `char`.

CHAR_MIN
Minimum value for an object of type `char`.

INT_MIN
Minimum value for `int` data type.

INT_MAX
Maximum value for `int` data type.

LONG_MAX
Maximum value for `long int` data type.

LONG_MIN
Minimum value for `long int` data type.

MB_LEN_MAX
Maximum number of bytes in a multibyte character.

SCHAR_MIN
Minimum value for `signed char` data type.

SCHAR_MAX
Maximum value for `signed char` data type.

SHRT_MAX
Maximum value for `short int` data type.

SHRT_MIN
Minimum value for `short int` data type.

UCHAR_MAX
Maximum value for `unsigned char` data type.

UINT_MAX
Maximum value for `unsigned int` data type.

ULONG_MAX
Maximum value for `unsigned long int` data type.

USHRT_MAX
Maximum value for `unsigned short int` data type.

POSIX Extensions

Macros defined are:

_POSIX_ARG_MAX
Total number of bytes, including environment data, for one of the exec functions.

_POSIX_CHILD_MAX
For each real user ID, number of simultaneous processes.

_POSIX_LINK_MAX
A file's link count value.

_POSIX_MAX_CANON
A terminal canonical input queue's size in bytes.

_POSIX_MAX_INPUT
The buffer size for a terminal input queue in bytes.

_POSIX_NAME_MAX
Character length of a file name.

_POSIX_NGROUPS_MAX
Number of simultaneous supplementary group IDs for each process.

_POSIX_OPEN_MAX
Number of files that can be open for one process simultaneously.

_POSIX_PATH_MAX
Character length of a pathname.

_POSIX_PIPE_BUF
Number of bytes that can be written to a pipe without being interrupted.

The following macro constants are undefined (with `#undef`) by this header file:

- CHILD_MAX
- LINK_MAX
- MAX_INPUT
- MAX_CANON
- NAME_MAX

locale.h

Functions in this header file tailor the international output environment for functions that control character handling, string collation, date and time formatting, and numeric editing. This header file enables the output of a program to be customized for a specific nationality.

Only the `"C"` locale is implemented in CONVEX C.

ANSI C

The structure defined is:

lconv
Contains the characters that can be changed in each locale.

The macros defined in this file represent categories that are used by the `setlocale` function. They are:

LC_ALL
A composite of all the other categories.

LC_COLLATE
This category impacts the comparison function used by the functions `strcoll` and `strxfrm`.

LC_CTYPE
This category impacts the following functions:

- `isalnum`
- `isalpha`
- `iscntrl`
- `isdigit`
- `isgraph`
- `islower`
- `isprint`
- `ispunct`
- `isspace`
- `isupper`
- `isxdigit`

The function-like macros of the same name are not impacted by this category. Therefore, it is necessary to undefine a function-like macro before the function that is affected by LC_CTYPE can be used.

LC_MONETARY
This category impacts information returned by the `localeconv` function regarding monetary characters.

LC_NUMERIC
This category impacts the decimal point character used by formatted input and output functions, string conversion functions, and non-monetary formatting information returned by the `localeconv` function.

LC_TIME
This category impacts the `strftime` function.

Functions declared in this header file are:

struct lconv *localeconv(void)
Returns a pointer to a structure of current locale information. The structure is in the static storage class and cannot be modified by the application.

char *setlocale(int category, const char *locale)
Changes or queries a program's current locale. `category` is a category macro defined in this header file. The `locale` argument may three values:

- `""` — sets the category to the locale of the minimal C translation.
- `"C"` — sets the category to the locale of the minimal C translation.
- NULL — queries the system for the current locale of the specified category.

math.h

This header file declares math functions.

Math function errors are grouped into two categories: domain errors (EDOM) and range errors (ERANGE). Domain errors occur when the size of an argument causes significant inaccuracy in the function result. When a domain error occurs, the value of the macro EDOM is stored in `errno`, an external identifier declared in errno.h.

Range errors result when the computed value cannot be represented within the machine's precision. When a range error occurs, the value of the macro ERANGE is stored in `errno`.

> **CAUTION**
>
> Domain and range errors that occur in the extended (default) compatibility mode may not change the `errno` identifier. To force these errors to change the value of `errno`, recompile files that contain math functions and include `-D__NO_INLINE_MATH` on the command line. Refer to the Chapter 9, "CONVEX C Intrinsics," for more information.
>
> Domain and range errors that occur in the standard (`-std`) and strict (`-str`) compatibility modes do affect the `errno` identifier.

ANSI C

The macro defined is:

HUGE_VAL
Largest positive double precision value. It is not a constant expression and cannot be evaluated by the preprocessor.

Functions declared in this header file are listed below. If the result of the function overflows, the function returns `HUGE_VAL`; the sign is the same as that of the correct function result except for the tan function. The function returns zero when an underflow occurs. `errno` is set to the value of `ERANGE` only when an overflow occurs.

ANSI C math functions are:

double acos(double x)
Returns arccosine. The argument must be in the range [-1,+1].

double asin(double x)
Returns arcsine. The argument must be in the range [-1,+1].

double atan(double x)
Returns arctangent.

double atan2(double numer, double denom)
Returns arctangent of `numer/denom`. Both arguments may not be zero.

double ceil(double x)
Returns smallest integer not less than the argument, $\lceil x \rceil$.

double cos(double x)
Returns cosine of the argument that is measured in radians.

double cosh(double x)
Returns hyperbolic cosine.

double exp(double x)
Returns exponential, e^x.

double fabs(double x)
Returns absolute value, $|x|$.

double floor(double x)
Returns largest integer not greater than the argument, $\lfloor x \rfloor$.

double fmod(double numer, double denom)
Returns floating-point remainder of `numer/denom`.

double frexp(double number, int *pow)
Returns `f`, where $\frac{1}{2} \leq f$ or $f = 0$ and `number` $= f \times 2^{(*pow)}$. If `number` = 0, `*pow` and f are zero.

double ldexp(double x, int pow)
Returns $x \times 2^{pow}$.

double log(double x)
Returns natural logarithm of **x**.

double log10(double x)
Returns base-10 logarithm of **x**.

double modf(double number, double *i)
Splits `number` into a fraction, `f`, and an integer `*i`, where `f + (*i) = number`. The function returns `f`.

double pow(double x, double pow)
Returns x^{pow}.

double sin(double x)
Returns sine of the argument that is measured in radians.

double sinh(double x)
Returns hyperbolic sine.

double sqrt(double x)
Returns positive square root of the argument, $|\sqrt{x}|$.

double tan(double x)
Returns tangent of the argument that is measured in radians.

double tanh(double x)
Returns hyperbolic tangent.

Table 11-2 lists values returned by math functions when a domain error occurs.

Table 11-2: Math Function Return Values

Function	Domain Error Return Value
acos	acos(1)
asin	asin(1)
atan	pi/2
atan2	pi/2
cos	cos(0)
cosh	HUGE_VAL
sin	sin(0)
sinh	-HUGE_VAL
log	log(|x|)
log10	log10(|x|)
pow	pow(|x|)
sqrt	sqrt(|x|)

CONVEX Extensions

When a range or domain error occurs for the following functions, the global integer `errno` is set to the value of a math error. All the math errors are defined in errno.h. These errors are more specific than `EDOM` and `ERANGE`.

double atof(const char * str)
Returns the `double` representation of the first number contained in the string pointed to by `str`. This ANSI C function is declared in this file for convenience; the ANSI C standard places this function in stdlib.h.

double cabs(struct { double x, y;} z)
Returns `z`'s Euclidean length.

double dcvtid(double x)
Converts native mode input, `x`, into IEEE floating-point mode.

double gamma(double x)
Returns the log gamma of `x`.

double hypot(double x, double y)
Returns the Euclidean distance of `x` and `y`.

double idcvtd(double x)
Converts IEEE mode input, `x`, into native floating-point mode.

int ipow(int x, int pow)
Returns x^{pow}.

float ircvtr(float x)
Converts IEEE mode input, `x`, into native floating-point mode.

double j0(double x)
Bessel functions of the first kind (j1, order 0).

double j1(double x)
Bessel functions of the first kind (j1, order 1).

double jn(int n, double x)
Bessel functions of the first kind (j1, order n).

long long int lpow(long long x, long long pow)
Returns x^{pow}.

float rcvtir(float x)
Converts native mode input, `x`, into IEEE floating-point mode.

double y0(double x)
Bessel functions of the second kind (y1, order 0).

double y1(double x)
Bessel functions of the second kind (y1, order 1).

double yn(int n, double x)
Bessel functions of the second kind (y1, order n).

For descriptions of these functions, refer to their man pages.

setjmp.h

This header file declares two functions that can be used instead of the normal function call and return paradigm.

ANSI C

The type defined is:

jmp_buf
Declares an identifier that retains the context of the calling environment. It is used by the two functions declared in this header file.

Functions declared are:

int setjmp(jmp_buf buffer)
Save the calling environment context in `buffer`.

void longjmp(jmp_buf buffer, int retval)
Restore the environment context saved in `buffer`. Program execution resumes with the statement following the setjmp function associated with `buffer`. the setjmp function returns `retval` if it is nonzero; if `retval = 0`, the setjmp function returns 1.

POSIX Extensions

The type defined is:

sigjmp_buf
Declares a location in which to store the calling environment information.

Functions declared are:

int sigsetjmp(sigjmp_buf env, int savemask)
Saves the calling environment in `env` and if `savemask` is nonzero, saves the process's current signal mask as part of the calling environment.

void siglongjmp(sigjmp_buf, int)
Restores the calling environment saved in `env` and if the signal mask is saved in the calling environment, restores that as well.

signal.h

This header file declares the functions that control the behavior of a program when signals occur. It also defines macro constants that represent various signals.

ANSI C

The type defined is:

sig_atomic_t
No interrupts are recognized when a value is assigned to an object of this type. This type is useful for communication between the program and its signal handlers.

The functions are:

void (*signal(int sig, void (*func)(int)))(int)
Define the behavior of the program when a particular signal is received.

int raise(int sig)
Sends a signal to the executing program.

A function-like macro is:

int raise(sig)

CONVEX Extensions

Refer to the header file for a list of the signals. Basic categories are:

- Traps
- Floating-point exceptions
- Bus errors
- Segment errors

Structures defined are:

sigaction
An aggregate of a signal handler, a bit mask, and some flags.

sigvec
Overlaid by struct sigaction.

sigcontext
Contains information pushed on the stack when a signal is delivered. It is made available to the handler to allow it to properly restore state if a non-standard exit is performed.

A function-like macro is:

int sigmask(number)
Creates a mask for the bit indicated by `number`. For example, the value of sigmask(5) is 16.

A function is:

int (*signal(int sig, void (*func)(int sig, int subcode, struct sigcontext *scp)))(int)
Defines the behavior of the program when a particular signal is received. The difference between this function and the ANSI C signal function is that the function `func` accepts three parameters, while the ANSI C signal handler accepts only one parameter.

POSIX Extensions

The structures defined are:

sigaction
An aggregate of a signal handler, a bit mask, and some flags.

sigvec
Overlaid by struct sigaction.

The following three macros defined are used as the first argument to the `sigprocmask` macro. They determine how the signal set of a process is changed.

SIG_BLOCK
Resulting signal set is a combination of the current signal set and the signal set pointed to by another argument to the function.

SIG_UNBLOCK
Resulting signal set is an intersection of the current signal set and the complement of the signal set pointed to by another argument to the function.

SIG_SETMASK
Resulting signal set is the signal set pointed to by another argument to the function.

Functions declared are:

int kill(pid_t pid, int sig)
Sends signal `sig` to the process specified by `pid`. Returns 0 if the function is successful; otherwise it returns -1.

int sigaction(int sig, struct sigaction *act, struct sigaction *oact)
Customizes or examines the action associated with a particular signal. `act` is a pointer to the new action; `oact` is used to store the old action if it is not NULL. If `act` is NULL, the signal handling is not changed. Returns 0 if the function is successful; otherwise, -1. Returns 0 if the function is successful; otherwise, -1.

int sigaddset(sigset_t *set, int signo)
Adds the individual signal specified by the value of `signo` to the signal set pointed to by `set`. Returns 0 if the function is successful; otherwise, -1.

int sigdelset(sigset_t *set, int signo)
Removes the individual signal specified by the value of `signo` from the signal set pointed to by `set`. If the function detects no errors, 0 is returned; otherwise; -1 is returned.

int sigemptyset(sigset_t *set)
Excludes all POSIX signals from the initialization of the signal set pointed to by `set`. If the function detects no errors, 0 is returned; otherwise -1 is returned.

int sigfillset(sigset_t *set)
Includes all POSIX signals in the initialization of the signal set pointed to by `set`. If the function detects no errors, 0 is returned; otherwise -1 is returned.

int sigismember(sigset_t *set, int signo)
Tests whether the set pointed to by `set` contains the signal specified by the value of `signo`. Returns 1 if the signal is a member of the set, and 0 if it is not. If an error occurs, -1 is returned.

int sigpending(sigset_t *set)
`set` is the storage location for the set of signals that are blocked from delivery and pending for the calling process. Returns 0 if it successful, and -1 if there is an error.

int sigprocmask(int how, sigset_t *set, sigset_t *oset)
Examines or changes the calling process's signal mask. `how` is one of the three macros described previously; `set` lists the modifications required as specified by `how`; and `oset` is the resulting signal set. If `set` is NULL no modifications take place, and `oset` contains the current signal set. Returns 0 if function is successful; -1 otherwise.

int sigsuspend(sigset_t *sigmask)
The set of signals pointed to by `sigmask` replaces a process's current signal mask. The process is suspended until it receives a signal that terminates it or a signal that forces it to execute a signal-catching function. Returns a -1 if an error occurs.

stdarg.h

The contents of this file are used to access parameters of functions that may have a variable number of arguments.

ANSI C

The type defined is:

va_list
Declares a variable argument list structure.

Function-like macros are:

***type* *va_arg(va_list list, *type*)**
Returns the next parameter in the argument list. The second argument is the type of the next parameter.

void va_end(va_list list)
Releases access to the structure defined by `va_list`.

void va_start(va_list list, <LastParam>)
initializes the data structure `list`. `<LastParam>` is the last nonvariable parameter in the function argument list.

stddef.h

ANSI C

Types defined are:

ptrdiff_t
Type resulting from the subtraction of two pointers.

size_t
Type returned by the `sizeof` macro operator.

wchar_t
Data type for wide characters.

The macro defined is:

NULL
Null pointer constant.

Function-like macro is:

size_t offsetof(structure, field)
Returns the offset, in bytes, of a structure member from the base address of its structure.

stdio.h

This header file declares types, macros, and functions that are used for input and output.

ANSI C

Macros defined are:

BUFSIZ
Buffer size for setbuf.

EOF
Value returned by functions that indicates the end of a file.

FILENAME_MAX
Maximum number of characters permitted in a file name.

FOPEN_MAX
Minimum number of files that can be open simultaneously.

_IOFBF
Parameter of `setvbuf` function that indicates full buffering for input/output.

_IOLBF
Parameter of `setvbug` function that indicates line buffering for input/output.

_IONBF
Parameter of `setvbug`etvbuf function that indicates no buffering for input/output.

L_tmpnam
Maximum length of a file name that is returned by the `tmpnam` function.

NULL
Null pointer constant.

SEEK_CUR
Parameter of the `fseek` function that sets the file reference point at the beginning of the file.

SEEK_END
Parameter of the `fseek` function that sets the file reference point at the end of the file.

SEEK_SET
Parameter of the `fseek` function that sets the file reference point at the current file position.

stderr
File pointer for standard error stream.

stdin
File pointer for standard input stream.

stdout
File pointer for standard output stream.

TMP_MAX
Minimum number of unique file names that can be produced by the tmpnam function.

The types defined are:

FILE
Object that records all information needed to access a file.

fpos_t
Object that retains all information required to specify uniquely a file position.

size_t
Type returned by the `sizeof` macro operator.

Operations On Files

int remove(const char *filename)
Discards the link between a file name and its file contents. Returns a nonzero value when an error occurs.

int rename(const char *old, const char *new)
Associates a new file name with a file. The link between `*old` and the contents of the file is removed. Returns a nonzero when an error occurs.

FILE *tmpfile(void)
Creates a temporary binary file. When the program terminates, the file is removed automatically. NULL is returned if an error occurs.

char *tmpnam(char *filename)
Returns a valid, unique file name.

File Access Functions

int fclose(FILE *file)
Flushes the buffer associated with `file`, then closes it. Returns EOF if an error occurs.

int fflush(FILE *file)
Flushes the buffer associated with `file`. Returns EOF if a write error occurs.

FILE *fopen(const char *filename, const char *mode)
Initiates access to a file. `mode` determines the type of access. Returns a pointer to the file structure or NULL if an error occurs.

FILE *freopen(const char *filename, const char *mode, FILE *fp)
Associates a new file with an existing file pointer. Any file already associated with the stream is closed. Access to the file is specified by `mode`. Returns NULL if an error occurs.

void setbuf(FILE *fp, char *buffer)
Establishes full buffering for a stream with `buffer` or causes input and output with a file pointer, `fp`, to be unbuffered.

int setvbuf(FILE *fp, char *buffer, int buf_type, size_t size)
Tailors the type of buffering associated with a file pointer to fully buffered, line buffered, or unbuffered. The third parameter selects the buffer type. It can have one of three values: _IOFBF, _IOLBF, or _IONBF. It also permits a user-defined buffer, specified by `buffer` and `size`. Returns a nonzero value if an error occurs.

Formatted Input/Output Functions

int fprintf(FILE *fp, const char *format, ...)
Formats text, as specified by `format` and writes the text to the file pointed to by `fp`. Returns the number of characters written.

int fscanf(FILE *fp, const char *format, ...)
Reads input from the file pointed to by `fp` using `format` and places the input data in the addresses specified by additional arguments. Returns EOF if no data is read; otherwise, it returns the number of conversions that were performed.

int printf(const char *format, ...)
Performs the same function as `fprintf` but writes to `stdout`.

int scanf(const char *format, ...)
Performs the same function as `fscanf` but reads from `stdin`.

int sprintf(char *array, const char *format, ...)
Performs the same function as `fprintf`, except the formatted text is written into the array pointed to by `array`. The null character is appended to the formatted text.

int sscanf(const char *array, const char *format, ...)
Performs the same function as fscanf, except the input is obtained from the array pointed to by `array`.

int vfprintf(FILE *stream, const char *format, va_list arg)
Performs the same function as `fprintf`. `arg` specifies the variable argument list. `arg` must be initialized with `va_start` before `vfprintf` is called. The header file stdarg.h must be included when this function is used.

int vprintf(const char *format, va_list arg)
Performs the same function as `printf`. `arg` specifies the variable argument list. `arg` must be initialized with `va_start` before `vprintf` is called. The header file stdarg.h must be included when this function is used.

int vsprintf(char *array, const char *format, va_list arg)
Performs the same function as `sprintf`. `arg` specifies the variable argument list. `arg` must be initialized with `va_start` before `vsprintf` is called. The header file stdarg.h must be included when this function is used.

Character Input and Output Functions

int fgetc(FILE *fp)
Returns the next character from the file pointed to by `fp`. If the end of the file is encountered or an error occurs, EOF is returned.

char *fgets(char *str, int n, FILE *fp)
Reads `n`-1 characters, or up to a newline character, from the file pointed to by `fp` into the string pointed to by `str`.

int fputc(int ch, FILE *fp)
Outputs a character to the file pointed to by `fp`. `fputc` performs the same function as putc, except that `fputc` is a function, whereas `putc` is a macro.

int fputs(const char *str, FILE *fp)
Copies the null-terminated string `str` to the file pointed to by `fp`. Returns EOF if a write error occurs.

int getc(FILE *fp)
Returns the next character from the file pointed to by `fp`, or EOF if the end of the file is encountered or an error occurs.

int getchar(void)
Returns the next character from `stdin`, or EOF if the end of the file is encountered or an error occurs.

char *gets(char *str)
Reads a string from the standard input `stdin`. Reads up to a newline character or the end of the file. A null pointer is returned if an error occurs.

int putc(int ch, FILE *fp)
Outputs a character to the file pointed to by `fp`. Returns EOF if a write error occurs.

int putchar(int ch)
Outputs a character to the standard output, `stdout`. Returns EOF if a write error occurs.

int puts(const char *str)
Writes the null-terminated string pointed to by `str` to the standard output, `stdout`, and appends a newline character. Returns EOF if an error occurs.

int ungetc(int ch, FILE *fp)
Pushes character, `ch`, back into an input buffer associated with the file pointer `fp`. Returns EOF if an error occurs.

Direct Input and Output Functions

size_t fread(void *ptr, size_t size, size_t nmemb, FILE *fp)
Reads a block of data from the file associated with `fp`. `ptr` is the base address of the array for the data; `nmemb` is the number of elements in the array; and `size` is the number of bytes required for each element in the array. Returns the number of array elements actually read.

size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
Writes a block of data to the file associated with `fp`. `ptr` is the base address of the data to write; `nmemb` is the number of items to write; and `size` is the number of bytes that represents each item. Returns the number of items actually written.

File Positioning Functions

int fgetpos(FILE *fp, fpos_t *pos)
pos is set equal to the current file position indicator of the file pointed to by fp. If an error occurs, a nonzero value is returned, and errno is set equal to one of the values in Table 11-3.

Table 11-3: errno values of fgetpos, fsetpos, and ftell

Value Returned	Error Condition
EBADF	file is not open
ESPIPE	file name is associated with a pipe or a socket
EINVAL	file position is negative

int fseek(FILE *fp, long int offset, int ref_pt)
Sets the file position of the file pointed to by fp to the position indicated by ref_pt and offset. ref_pt can be one of three values: SEEK_CUR, the current file position; SEEK_SET, the beginning of the file; and SEEK_END, the end of the file. offset is the number of bytes from the reference point. Returns a nonzero value when a domain error occurs.

int fsetpos(FILE *fp, const fpos_t *pos)
Sets file position of file pointed to by fp equal to value stored in pos that originated in a prior call to fgetpos. Sets the file position indicator for stream according to the value of the object pointed to by pos, which is be a value obtained from an earlier call to fgetpos on the same file.

long ftell(FILE *fp)
Returns the offset of the current file position of the file pointed to by fp from the beginning of the file. The value is measured in bytes. If an error occurs, it returns -1L and sets errno equal to a value listed in Table 11-2.

void rewind(FILE *fp)
Sets the file position indicator to the beginning of the file pointed to by fp.

Error-handling Functions

void clearerr(FILE *fp)
clears end-of-file and error indicators for the file associated with fp.

int feof(FILE *fp)
Returns a nonzero value if the end-of-file indicator for the file associated with fp is set.

int ferror(FILE *fp)
Returns a nonzero value if the error indicator of the file associated with fp is set.

void perror(const char *str)
Writes the error message associated with the value stored in `errno` to `stderr`. This message is preceded by the string pointed to by `str`. Tables E-8 through E-15 list the error messages associated with each valid value of `errno`.

Routines in the following list are available as function-like macros and as functions:

- getc
- getchar
- putc
- putchar
- feof
- ferror

CONVEX Extensions

The function declared is:

FILE *popen(char *str, char *mode)
Creates a pipe between the calling process and the command to be executed, which is pointed to by `str`. `mode` can be `"r"` for reading or `"w"` for writing. The value returned is a pointer that can be used to write to the standard input of the command or read from its standard output. If an error occurs, `NULL` is returned.

int pclose(FILE *pp)
Waits for the process associated with the pipe pointer, `pp`, to terminate and returns the exit status of the command. Returns -1 on a domain error.

POSIX Extensions

Two macros are defined:

L_cuserid
Maximum length of the controlling user ID including the terminating null byte.

L_ctermid
Maximum length of the controlling terminal ID including the terminating null byte.

The following functions are declared:

int *fdopen(int fildes, const char *type)
Associates a file with a file descriptor.

int fileno(const FILE *fp)
Returns the integer file descriptor associated with `fp`. This is also available as a function-like macro.

char *cuserid(char *str)
Returns a name associated with the effective user ID of the process.

stdlib.h

This header file contains declarations for some general utilities.

ANSI C

The macros defined are:

EXIT_FAILURE
Argument of `exit` that indicates abnormal termination status.

EXIT_SUCCESS
Argument of `exit` that indicates normal termination status.

NULL
Null pointer constant.

MB_CUR_MAX
Number of bytes required to represent a multibyte character in the current locale.

RAND_MAX
The maximum integer returned by rand.

The types defined in this file are:

size_t
Result type of the `sizeof` operator.

wchar_t
Data type for wide characters.

div_t
Return type of the div function.

ldiv_t
Return type of the ldiv function.

String Conversion Functions

double atof(const char *nptr)
Returns the `double` representation of the initial portion of the string pointed to by `nptr`.

int atoi(const char *nptr)
Returns the `int` representation of the initial portion of the string pointed to by `nptr`.

int atol(const char *nptr)
Returns the `long int` representation of the initial portion of the string pointed to by `nptr`.

double strtod(const char *nptr, char **endptr)
Returns the `double` representation of the initial portion of the string pointed to by `nptr`. Returns 0 if conversion is unsuccessful. Returns HUGE_VAL and sets `errno` to ERANGE if an overflow occurs. Returns 0 and sets `errno` to ERANGE if an underflow occurs.

long strtol(const char *nptr, char **endptr, int base)
Returns the `long int` representation of the initial portion of the string pointed to by `nptr`. Returns 0 if conversion is unsuccessful. If a range error occurs it sets `errno` to ERANGE and returns LONG_MIN if the sign of the answer is negative and LONG_MAX if the sign is positive.

unsigned long strtoul(const char *nptr, char **endptr, int base)
Returns the `unsigned long int` representation of the initial portion of the string pointed to by `nptr`. Returns 0 if conversion is unsuccessful. If a range error occurs, `errno` is set to ERANGE and ULONG_MAX is returned.

Pseudo-random Sequence Generation Functions

int rand(void)
Returns a nonnegative integer less than or equal to `RAND_MAX`.

void srand(unsigned int seed)
Reinitialize the sequence of pseudo-random integers.

Memory Management Functions

void *calloc(size_t nmemb, size_t size)
Allocates a block of memory for `nmemb` objects, each of size `size`.

void free(void *pm)
Deallocates the memory pointed to by `pm`.

void *malloc(size_t size)
Returns a pointer to a memory block that has `size` bytes. Returns a null pointer if sufficient memory is not available.

void *realloc(void *ptr, size_t size)
Changes the memory size originally allocated to `ptr`. The new memory size requirement is specified by `size`. Returns the new memory pointer or a null pointer if an error occurs.

Communication with the Environment Functions

void abort(void)
Causes program to terminate abnormally if the signal SIGABRT is not cleared by a signal handler. Does not return program execution to its caller.

int atexit(void (*func)(void))
The function pointed to by `func` is registered. Functions that are registered are executed when a program ends. The `func` function
is called without arguments at normal program termination. The `atexit` function returns a non-zero value if the `func` function could not be registered.

void exit(int status)
Causes program to terminate normally.

char *getenv(const char *name)
Searches environment list for a string of the form `name=`*value*. Returns a pointer to the string *value* if such a string is present. If such a string is not present, getenv returns the NULL pointer.

int system(const char *string)
Issues a shell command. The current process waits until the shell has completes the command indicated by `string`, then returns the exit status of the shell.

Searching and Sorting Functions

void *bsearch(const void *key, const void *base, size_t nmemb,
size_t size, int (*compare)(const void *, const void *))
Searches for `key` in the array pointed to by `base` using the function `compare`. Each array element has `size` bytes. Returns a pointer to the matching element of the array if it exists; otherwise it returns a null pointer.

void qsort(void *base, size_t nmemb, size_t size,
int (*compare)(const void *, const void *))
Provides a quick-sort utility. `base` is a pointer to the base of the data; `nmemb` is the number of elements; `size` is the width of an element in bytes; `compare` is the name of the routine that compares two elements in the array.

Integer Arithmetic Functions

int abs(int j)
Returns the absolute value of `j`.

div_t div(int numer, int denom)
Returns the quotient and remainder of the arguments. The structure returned has two members, `quot` and `rem`.

long labs(long j)
Returns the `long` representation of the absolute value of `j`.

ldiv_t ldiv(long numer, long denom)
Returns the `long` representation of the quotient and remainder of the arguments. The return value has two members, `quot` and `rem`.

Multibyte Character Functions

These functions are of limited usefulness because CONVEX C supports only the `"C"` locale. If `*str` is NULL in any of the following functions, 0 is returned indicating that no state-dependent encodings exist.

int mblen(const char *str, size_t n)
Returns the number of bytes, up to `n`, of the multibyte character pointed to by `str`. This is always 1 unless `n` == 0 or `*str` == NULL.

int mbtowc(wchar_t *pwc, const char *str, size_t n)
Because all multibyte characters are length 1, the first character pointed to by `*str` is copied to `*pwc` and `1` is returned. However, if `n` = 0, no conversion is performed and `-1` is returned.

int wctomb(char *str, wchar_t wchar)
Converts the multibyte character code stored in the object `wchar` into a multibyte character whose base address is `str`. Returns 1 because all multibyte characters are of length 1.

Multibyte String Functions

These functions are of limited usefulness because CONVEX C supports only the `"C"` locale. Conversions between multibyte strings and wide characters are trivial because no special encodings are supplied by CONVEX. If `*str` is NULL in any of the following functions, 0 is returned indicating that no state dependent encodings exist.

size_t mbstowcs(wchar_t *pwcs, const char *str, size_t n)
Copies `n` bytes, or until NULL is encountered, of the multibyte string pointed to by `*str` to the wide character string pointed to by `*pwcs`. Returns the number of multibyte characters copied.

size_t wcstombs(char *str, const wchar_t *wcs, size_t n)
Copies an array of `n` or fewer multibyte character encodings into a multibyte character string. Returns one less than the number of characters copied.

string.h

String-handling functions are used to manipulate character arrays and blocks of memory.

ANSI C

One macro is defined:

NULL
Null pointer constant.

One type is defined:

size_t
Type returned by the `sizeof` macro operator.

Copying Functions

void *memcpy(void *str1, const void *str2, size_t n)
Copies the array of type `char` pointed to by `str2` into the array pointed to by `str1`. `n` or fewer elements are copied. If the arrays overlap, the behavior is undefined. Returns `str1`.

void *memmove(void *str1, const void *str2, size_t n)
Copies the array of type `char` pointed to by `str2` into the array pointed to by `str1`. `n` or fewer elements are copied. The arrays may overlap. Returns `str1`.

char *strcpy(char *str1, const char *str2)
Copies the null-terminated string pointed to by `str2` into the string pointed to by `str1`. If the strings overlap, the behavior is undefined. Returns `str1`.

char *strncpy(char *str1, const char *str2, size_t n)
Copies exactly `n` characters from string pointed to by `str2` into the string pointed to by `str1`, truncating or null-padding if necessary. If the strings overlap, the behavior is undefined. Returns `str1`.

Concatenation Functions

char *strcat(char *str1, const char *str2)
Concatenates the string pointed to by `str2` at the end of the string pointed to by `str1`. The null-terminating character of `str1` is overwritten by the first character of `str2`. If the strings overlap, the behavior is undefined. Returns `str1`.

char *strncat(char *str1, const char *str2, size_t n)
Concatenates `n` or fewer characters of the string pointed to by `str2` at the end of the string pointed to by `str1`. The null-terminating character of `str1` is overwritten by the first character of `str2`. If the strings overlap, the behavior is undefined. Returns `str1`.

Comparison Functions

int memcmp(const void *buf1, const void *buf2, size_t n)
Compares the first `n` bytes of `buf1` and `buf2`. Each byte is considered to be an `unsigned char`. Returns:

— a positive integer if `buf1` is greater than `buf2`.
— zero if the two buffers are the same.
— a negative integer if `buf1` is less than `buf2`.

int strcmp(const char *str1, const char *str2)
Compares characters in two strings and returns:

— a positive integer if the string pointed to by `str1` is greater than the string pointed to by `str2`.
— zero if the two strings are the same.
— a negative integer if the string pointed to by `str1` is less than the string pointed to by `str2`.

int strcoll(const char *str1, const char *str2)
Compares characters in two strings. The comparison function is affected by the current locale category LC_COLLATE. Returns:

— a positive integer if the string pointed to by `str1` is greater than the string pointed to by `str2`.
— zero if the two strings are the same.
— a negative integer if the string pointed to by `str1` is less than the string pointed to by `str2`.

int strncmp(const char *str1, const char *str2, size_t n)
Compares the first `n` characters of two strings and returns:

— a positive integer if the string pointed to by `str1` is greater than the string pointed to by `str2`.
— zero if the first `n` characters of the two strings are the same.
— a negative integer if the string pointed to by `str1` is less than the string pointed to by `str2`.

size_t strxfrm(char *str1, const char *str2, size_t n)
Transforms sufficient characters in the string pointed to by `str2` such that the string pointed to by `str1` will have `n` or fewer characters. The transformation does not change the expected result of the `strcoll` function. Returns the number of characters in the transformed string.

Search Functions

void *memchr(const void *str, int ch, size_t n)
Returns a pointer to the first location of the character `ch` (interpreted as `unsigned char`) in the first `n` characters of the string pointed to by `str`. If no character is found, it returns a NULL pointer.

char *strchr(const char *str, int ch)
Returns a pointer to the first location of the character `ch` (interpreted as `char`) in the string pointed to by `str`. If no character is found, it returns NULL.

size_t strcspn(const char *str1, const char *nset)
Returns the length of the largest initial substring of the string pointed to by `str1`. The substing cannot contain any characters in the string pointed to by `nset`.

char *strpbrk(const char *str1, const char *set)
Returns a pointer to the first occurrence in the string pointed to by `str1` of any character in the string pointed to by `set`. Returns a pointer to such a character or a NULL pointer if a matching character is not found.

char *strrchr(const char *str, int ch)
Returns a pointer to the last location of the character `ch` (interpreted as `char`) in the string pointed to by `str`. If no character is found, it returns NULL.

size_t strspn(const char *str1, const char *set)
Returns the length of the largest initial substring of the string pointed to by `str1` consisting only of characters contained in the string pointed to by `set`.

char *strstr(const char *str1, const char *pattern)
Returns a pointer to the first substring that matches the string pointed to by `pattern` in the string pointed to by `str1`. Returns NULL if no such substring is found.

char *strtok(char *str, const char *delimiters)
Divides the string pointed to by `str` into tokens delimited by the characters in the string pointed to by `delimiters`. The first function call returns a pointer to the first token in `str`. Delimiters are overwritten by the NULL character. Subsequent calls to `strtok` must have NULL as the first argument; a pointer to the next token is returned. If a token is not found, the return value is NULL.

Miscellaneous Functions

void *memset(void *buf, int ch, size_t n)
Initializes the first `n` characters of the array pointed to by `buf` with the character `ch`.

char *strerror(int errno)
Returns a pointer to the error message associated with the error number `errno`. A list of the error messages associated with each value of `errno` is contained in Tables E-8 through E-15 This is also available as a function-like macro.

size_t strlen(const char *str)
Returns the number of characters in the string pointed to by `str`.

CONVEX Extensions

Functions declared are:

char *index(char *string, int ch)
Returns a pointer to the first occurrence of the character `ch` (interpreted as `char`) in the string pointed to by `string`. If no character is found, it returns NULL.

char *rindex(char *string, int ch)
Returns a pointer to the last occurrence of the character `ch` (interpreted as `char`) in the string pointed to by `string`. If no character is found, it returns NULL.

time.h

ANSI C

Macros defined in this header file are:

NULL
Null pointer constant.

CLOCKS_PER_SEC
The number of times the system clock increments for each second.

Types defined in this header file are:

size_t
The type returned by the `sizeof` macro operator.

clock_t
The type returned by the clock function.

time_t
The type returned by the time function.

One structure is defined:

tm
Members of this structure are:

tm_sec
Seconds after the minute.

tm_min
Minutes after the hour.

tm_hour
Hours since midnight.

tm_mday
Day of the month.

tm_mon
Months since January.

tm_year
Years since 1990.

tm_wday
Days since Sunday.

tm_yday
Days since January 1.

tm_isdst
Daylight savings time flag: positive if DST is in effect; 0 if DST is not in effect; negative if DST status is unknown.

Time Manipulation Functions

clock_t clock(void)
Returns the execution time for the current process.

double difftime(time_t time1, time_t time0)
Returns the difference in seconds between two times. This is also available as a function-like macro.

time_t mktime(struct tm *timeptr)
Converts the time in `tm` representation to time in `time_t` representation. Returns the converted value or -1 if the conversion cannot be performed. The structures are described above.

time_t time(time_t *timer)
Returns the current time in a `time_t` representation. If `timer` is not NULL, the value returned is also stored in `timer`. the `time_t` structure is described above.

Time Conversion Functions

char *asctime(const struct tm *timeptr)
Converts a time in the `tm` representation into a string in the format:
`Wed Apr 25 23:26:45 1962\n\0`
(The \n and \0 values are a carriage-return and null value, respectively.) Returns a pointer to the string.

char *ctime(const time_t *timer)
Converts the `time_t` representation pointed to by `timer` into an ASCII representation.

struct tm *gmtime(const time_t *timer)
Converts time pointed to by `timer` from the `time_t` structure to the `tm` structure.

struct tm *localtime(const time_t *timer)
Converts time returned by the time function, pointed to by `timer`, to a structure containing the broken-down time, correcting for time zone and if necessary, daylight savings time.

**size_t strftime(char *str, size_t n, const char *format,
const struct tm *time)**
Converts `time` into a string pointed to by `str` using the format specified by `format`. The number of characters produced cannot be greater than `n`. If `n` is exceeded, the contents of the string are undefined and 0 is returned; otherwise, the number of characters produced is returned.

POSIX Extensions

One macro is defined:

CLK_TCK
Number of clock ticks for each second.

One external identifier is declared:

tzname
An array of time zone names.

Chapter 12
Preprocessor Directives

You can increase program readability and simplify program maintenance by defining macros for constants or short pieces of code. For example, if some functions use π, it is better to define a macro with a meaningful name that contains the value for π rather than to use the number 3.14 throughout the source file. Thus, if greater precision is needed, the value for π need be changed in only one place in the source file.

The preprocessor also supports function-like macros. These macros are shorthand for one or more lines of source code that use one or more parameters. For example, if a set of three statements is used frequently throughout a source file, a macro may be defined that represents the three lines. Then, whenever the three lines are needed, only the macro must be written. This is explained in more detail in the sections below.

Another feature of the preprocessor is its support for conditional compilation. Conditional compilation is a technique used to restrict the source code to be compiled. For example, it may be used to remove parts of the code that produce debugging information that is not needed in the completed application. Conditional compilation also makes it easier to port source code between computer systems.

Preprocessor directives are indicated by a # symbol which may appear anywhere on a line as long as it is preceded only by white space and C comments. The directive must follow the # symbol on the same line with only blanks, horizontal tabs, or C comments separating the two. The directive only affects the line it is on; there are no multiline directives.

Preprocessor directives may have zero or more arguments.

#define

The syntax of this directive is:

`#define` *name definition*

The definition for this directive is terminated by the newline character. In its simplest form, this directive is used to define constants. For example, the following line provides a definition for π:

```
#define PI 3.14159265
```

After defining this constant, use only the identifier PI when the value of π is required.

When more precision is required, change the definition of PI to

```
#include <math.h>
#define PI (4.0 * atan(1.0))
```

Because the definition of a name is terminated by a newline character, it is not limited to one word. When used in this manner, **#define** performs simple text substitution.

A multiword definition is:

```
#define UCHAR unsigned char
```

Thus, this type of definition replaces commonly used pieces of code.

The '\' (backslash or continuation) character placed at the end of a line permits the definition of a name to extend over multiple lines. The '\' character must be followed by a newline; it cannot be followed by blanks, tabs, or comments. Lines connected with the '\' character are recognized by the preprocessor as a single line. For example, the following definition is considered to be on one line:

```
#define close_files fclose( file1 );\
    fclose( file2 );\
    fclose( file3 )
```

When the word **close_files** is seen in the source file in which this definition is visible, the preprocessor replaces it with the three C statements.

Finally, this directive may define a function-like macro that takes arguments. For example, the previous definition may be written as:

```
#define close_files( file1, file2, file3 )\
    fclose( file1 );\
    fclose( file2 );\
    fclose( file3 )
```

This permits macro definitions to be more flexible. A macro function that takes arguments must have the left parenthesis immediately follow the macro name; no intervening spaces are permitted.

Even though a function-like macro looks like a C function, there are differences:

- Function calls require more overhead; therefore, they may be slower.
- Function-like macros increase the length of a source file.
- Parameters in a macro may have unexpected side effects.

An example of a macro with an unexpected side effect is:

```
#define DIV2( quotient, divisor )\
    if( divisor >= 0 && divisor < sizeof( quotient ))\
        quotient >> divisor
```

This function yields correct results only if `divisor` is a power of two. This macro replaces a call to DIV2 with an integer division performed by a binary shift. It may produce incorrect results if its parameters are defined incorrectly:

```
num1 = 21;
num2 = 1;
result = DIV2( num1, num2++);
```

This macro function produces unexpected results because the second parameter is incremented each time it is used in the Boolean expression to which DIV2 expands. The macro expansion of the above code is:

```
num1 = 21;
num2 = 1;
result = if( num2++ >= 0 && num2++ < sizeof( num1 )) num1 >> num2++;
```

Each time `num2` is accessed, its value is incremented. This leads to an incorrect answer.

#include

The #include directive replaces the specified line with a specified file. The format of this directive is:

```
#include "filename"
```

or

```
#include <filename>
```

The directory search order for `#include` files enclosed in double quotes is:

1. The directory of the file that contains the `#include` directive
2. The directories specified by the `-I` command line option
3. The system include file directory, /usr/include

The directory search order for `#include` files delimited by angle brackets is:

1. The directories specified by the `-I` command line option
2. The system include file directory, /usr/include

#undef

The syntax of the `#undef` directive is:

```
#undef name
```

where *name* is an identifier that has been defined using the `#define` preprocessor directive or the `-D` command-line option.

Macro Operators

Two operators are useful in defining a macro: # and ##. The # operator converts a parameter to a string. This operator can be used only with formal arguments in the definition of a function-like macro. It converts the parameter to a string literal, as though the parameter was enclosed in double quotes. If the actual parameter contains white space, only the white space embedded within text is significant, and contiguous white space is reduced to one blank. Further, if the actual parameter contains symbols that are normally preceded by a backslash in a string literal, these symbols are inserted automatically into the resulting string literal.

For example, the following function-like macro can be used to open a file:

```
#define file_open( filename )\
    fopen( #filename, "r" )
```

This macro function could be used in the following manner:

```
FILE *fp;
fp = file_open( myfile );
```

The preprocessor expands this macro into, the following lines of code:

```
FILE *fp;
fp = fopen( "myfile", "r" );
```

A second operator that is sometimes used in macro definitions is ##, the token pasting operator. It creates one token from two tokens. For example, it may be used to perform operations on variables that are spelled similarly:

```
#define print_var( num ) \
    printf("var" #num " = %d\n", var##num )

int var1, var2, var3, var4;
print_var( 3 );
```

The macro expansion results in the following lines of code:

```
int var1, var2, var3, var4;
printf("var" "3" " = %d\n", var3 );
```

Conditional Compilation

Many directives can be used in conditional compilation. They are similar to their counterparts in C with one primary exception: the expression that the preprocessor evaluates must compute to a constant *prior* to program execution. Consequently, symbols in an expression are limited to macro names and integers.

The syntax of the `if` directive is:

```
#if expression
```

where *expression* is a combination of expressions constructed with C operators such as `&&`, `||`, and `!`.

An operator provided by the preprocessor is `defined`, which returns `1` if its operand has an existing macro definition. Macros are defined with the `#define` preprocessor directive or with the `-D` command-line option. For more information on the `-D` option, refer to Chapter 2, "Compiler Fundamentals."

If *expression* evaluates to a nonzero number, source code between `#if` and a following `#elif`, `#else`, or `#endif` directive remains in the source file.

The `#else` directive is the preprocessor counterpart to the `else` keyword; the `#elif` directive is the preprocessor counterpart to the `else if` keyword combination. The `#endif` directive terminates the `#if` directive.

For example, the following conditional code compiles different source code for different executing environments:

```
#define CONVEX            1
#define OTHER_COMPUTER    0
assert( !(CONVEX != 0 && OTHER_COMPUTER != 0) );

#if CONVEX
    long long j;
#elif OTHER_COMPUTER
    long j;
#endif
```

In this code, extensions to C are used, depending on the architecture of the execution environment. Note that the `assert` function is not executed until the program is run, while the conditional compilation occurs before runtime. If both CONVEX and OTHER_COMPUTER macro constants are nonzero, a diagnostic message is printed and the program halts. (Refer to the assert(3) man page for more information on this function-like macro.)

Two other conditional compilation directives are `#ifdef` and `#ifndef`. These are the same as `#if defined( )` and `#if !defined( )`, respectively.

#pragma

Pragmas provide additional information to the compiler; they also instruct the compiler to override conditions that automatically control vectorization or parallelization. Appendix B, "Pragmas," describes pragmas recognized by the compiler.

#error

The syntax of this directive is:

```
#error token token ...
```

where *token* can be a preprocessor macro or text. This directive displays an error message when the preprocessor is executing.

Instead of using the **assert** function, the following code may be used:

```
#if OTHER_COMPUTER != 0 && CONVEX != 0
#    error "illegal environment"
#endif
```

This code has the advantage of producing error messages during compilation time instead of execution time.

#line

The syntax of this directive is:

```
#line number [filename]
```

The `#line` directive associates a different line number and, optionally, a file name with a source file.

This directive is useful when the source file is generated by another program. For example, the origin of the source file may be a translator for another language, such as FORTRAN. This directive can be used to associate errors in the C source file with lines in the FORTRAN source file.

Chapter 13
The asm Statement

The `asm` statement is a CONVEX extension that is available only in the extended and backward-compatible modes. Prior versions of CONVEX C required the `-asm` compiler option to compile source files that contain `asm` statements; this option is *no longer necessary.*

Assembly-Language Statements

The CONVEX C compiler allows you to insert assembly-language statements into a C program using the `asm` statement. This statement has the form

asm ("*assembly_language_instruction*");

The compiler turns off global optimization and global register allocation when compiling source files that contain `asm` statements. Therefore, you must isolate functions that use the `asm` statement from functions that can be optimized.

Use the `asm` statement to embed assembly language statements in C code rather than writing an entire routine in assembly-language. For example:

```
void func(int interr[], int blast[])
{
    int i;

    asm ("dsi"); /* disable interrupts */
    for( i=0; i<10; i++ ){
        if( interr[i] )
            blast[i] = 1;
    }
    asm ("eni"); /* enable interrupts */
}
```

The available assembly-language instructions are described in the *CONVEX Assembly Language User's Guide* and the *CONVEX Architecture Reference.*

APPENDIX A
Data Types and Representations

This appendix describes data types that are supported by the CONVEX C compiler. These data types are listed below:

- Character
- Integer
- Enumeration
- Floating-point
- Pointer
- Void
- Union
- Structure
- Array

Data types that the compiler supports can be divided into three categories: those that ANSI C requires, those that are CONVEX extensions, and those that are accepted in the backward-compatible mode of the compiler.

Each discussion of a data type includes a description and a representation; examples clarify certain situations. Data types that are not accepted by an ANSI C compiler are noted. For each internal data representation, numbers at the top of the figure represent bit numbers; numbers at the bottom represent byte offsets. The least significant bit in the data representation is numbered 0.

Integral Types

char and int

The ANSI C compiler supports three integral data types: `char`, `short int`, and `int`. The bit length of each data type is shown in Table A-1.

Table A-1: Integral Type Bit Length

Data Type	Length
`char`	8-bit integer
`short int` or `short`	16-bit integer
`long int` or `int` or `long`	32-bit integer

By default, each data type is assumed to be a signed value. The range of numbers that these data types can represent is listed in Table A-2.

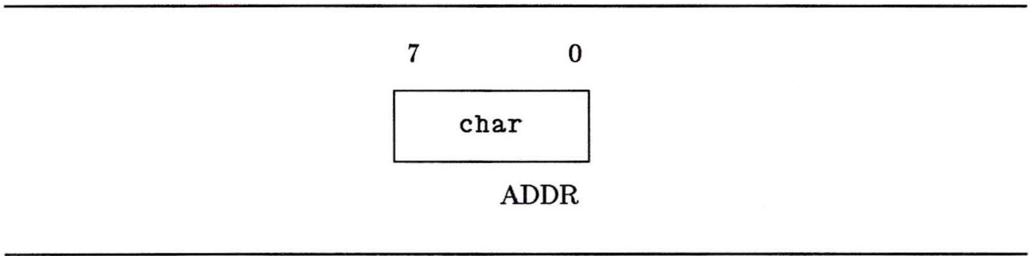
Table A-2: Integral Ranges

Data Type	Signed Range	Unsigned Range
`char`	-128 ... 127	0 ... 255
`short int` or `short`	-32,768 ... 32,767	0 ... 65,535
`long int` or `int`	-2,147,483,648 ... 2,147,483,647	0 ... 4,294,967,295

As ranges for the `char` data type imply, integer arithmetic can be performed on characters. The `char` representation contains a single character in the execution character set (ASCII).

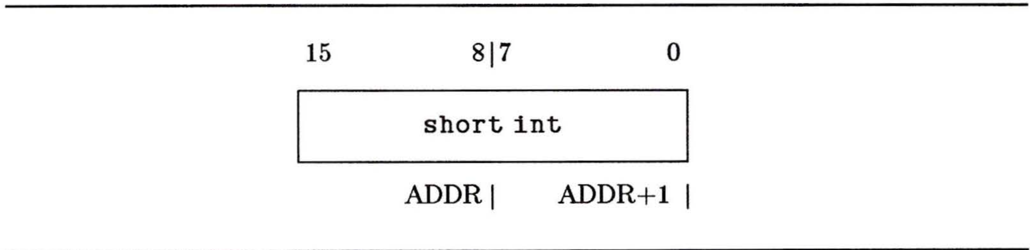
Figure A-1 illustrates the internal representation of the `char` data type.

Figure A–1: char Representation



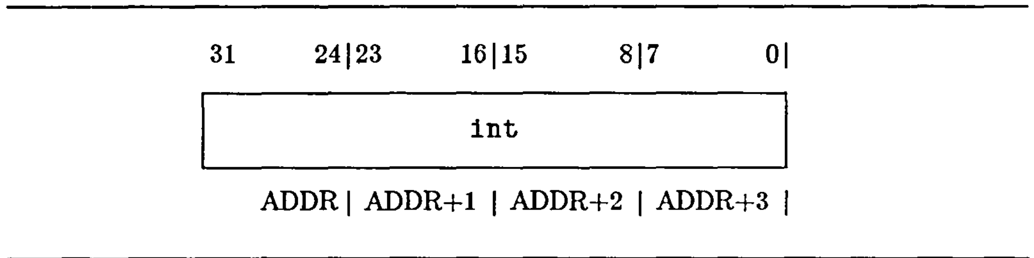
The internal representation of the `short int` data type is shown in Figure A-2.

Figure A–2: short int Representation



A 32-bit integer variable is declared as `int` (or `long int`) and can be unsigned or signed. Figure A-3 shows the `int` data type representation.

Figure A–3: int Representation



long long int

This data type is an integer type that is implemented in 64 bits. The range for the `long long` data type is shown in Table A-3.

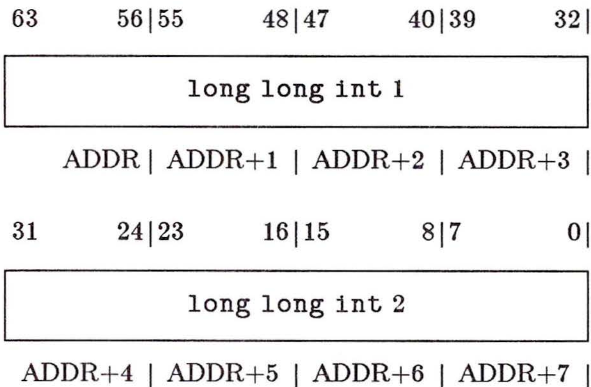
Table A-3: long long data type range

Format	long long or long long int
signed	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807
unsigned	0 ... 18,446,744,073,709,551,615

> **Note**
>
> The `long long` data type is a CONVEX extension to the ANSI C standard. It must not be used by programs that will be ported to other compilers. This data type is accepted in the backward-compatible and extended modes of the compiler.

Figure A-4 shows the `long long` data representation.

Figure A–4: long long Representation



In the backward-compatible mode `long long` integers cannot be passed as arguments to functions that do not expect to receive them. In general, `char` and `short` values can be passed to routines that expect `int` or `long` values. CONVEX C converts 8- and 16-bit quantities to a 32-bit format before pushing them onto the runtime stack. Because 64-bit values are not truncated, improper stack alignment results when `long long int` values are passed to routines that expect `int` arguments.

Enumerated Types

An enumerated data type is a user-defined integral data type. Declare enumerated scalar data as `enum`. For example, you might declare the enumerated data type `color` as:

```
enum color { red, blue, green } hue;
```

In this example, the variable `hue` could be only one of the values (`red`, `blue`, or `green`) at any given time.

Internally, `enum` values are stored as integer representations. By default, the first enumerated value (`red` in the example above) is stored with the ordinal value of zero. Subsequent enumerated values are represented by sequential integer values. In the example shown above, the value of `blue` is 1 and `green` is 2.

Default ordinal values are overridden when they are followed by an equal sign and a new ordinal (for example: `enum color {red=10, blue=20,green=30}`). Because the `enum` data type is implemented as a `signed int`, the range of ordinal values available for use is the same as that used by a `signed int` in Table A-2.

Floating-Point Types

The ANSI C compiler supports three floating-point data types: `float`, `double`, and `long double`. The bit length of each of these data types is listed in Table A-4.

Table A-4: Floating-Point Bit Length

Data Type	Length
`float`	32-bit floating-point
`double` and `long double`	64-bit floating-point

Floating-point data can be represented in either CONVEX native format or IEEE format. To process floating-point data in IEEE mode, you must specify the correct option (Refer to Chapter 2, "Compiler Fundamentals."), and your machine must have IEEE support hardware.

Note	CONVEX hardware and runtime libraries support only the processing of data encoded in IEEE format and do not conform to the IEEE 754 specifications for arithmetic.

The range of values provided with each format is shown in Tables A-5 and A-6.

Table A-5: Native and IEEE Floating-Point Ranges

Format	float
Native	$2.9387359 \times 10^{-39}$ ... $1.7014117 \times 10^{+38}$
IEEE	$1.1754944 \times 10^{-38}$ ... $3.4028235 \times 10^{+38}$

Table A-6: Native and IEEE Floating-Point Ranges

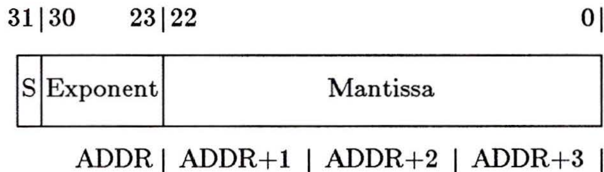
Format	double and long double
Native	$5.562684646268003 \times 10^{-309}$ ... $8.988465674311584 \times 10^{+307}$
IEEE	$2.225073858507201 \times 10^{-308}$ ... $1.797693134862317 \times 10^{+308}$

Floating-Point Representation: float

Single-precision (32-bit) floating-point variables are declared with the `float` keyword.

Positioning of the sign, exponent, and mantissa apply to native and IEEE formats; figure A-5 illustrates the internal representation of single-precision floating-point data.

Figure A–5: Single-Precision Floating Representation



Single-Precision Native

In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 128. That is, 128 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the left of the implicit 1 bit.

Single-Precision IEEE

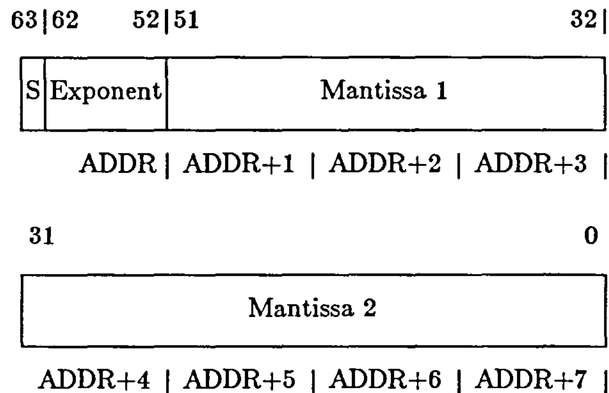
In the internal representation, the sign bit (S) is 0 for a positive number and 1 for a negative number. The exponent is an 8-bit binary field with a bias of 127; that is, 127 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 22. The binary point is to the right of the implicit 1 bit.

Floating-Point Representation: double, long double

Double-precision (64-bit) floating-point variables are declared with the `double` or `long double` keyword and can be represented in either native format or IEEE format. To process floating-point data in IEEE mode, the machine must have IEEE support hardware.

Figure A-6 shows the internal representation of double-precision floating-point data. The position of the sign, exponent, and mantissa apply to native and IEEE formats; particulars of each format are described in the following sections:

Figure A–6: Double-Precision Floating Representation



Double-Precision Native

In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1024; that is, 1024 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the left of the implicit 1 bit.

Double-Precision IEEE

In the internal representation, the sign (S) bit is 0 for a positive number and 1 for a negative number. The exponent is an 11-bit binary field with a bias of 1023; that is, 1023 must be subtracted from the exponent to give the actual power of 2. The mantissa is the fractional portion of the number and has an implicit 1 bit to the left of bit position 51. The binary point is to the right of the implicit 1 bit.

Floating-Point Type: long float

> **Note**
>
> The `long float` data type is a floating-point type supported only by the backward-compatible mode of CONVEX C. It should not be used by programs that will be ported to other computer systems.

This type is synonymous with the **double** and **long double** data types. The range for this type is detailed in Table A-7.

Table A-7: Native and IEEE long float Range

Format	long float
Native	$5.562684646268003 \times 10^{-309}$ ... $8.988465674311584 \times 10^{+307}$
IEEE	$2.225073858507201 \times 10^{-308}$ ... $1.797693134862317 \times 10^{+308}$

Pointer Data Type

A pointer is a variable that contains a 32-bit address. An asterisk is used to declare a pointer. For example, the declaration

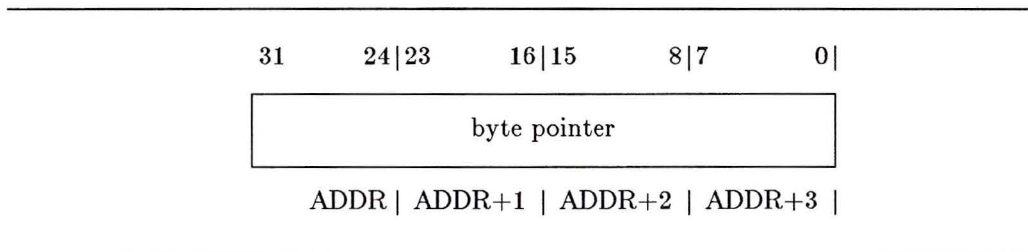
```
char *cp;
```

designates a pointer named `cp` that can be assigned the address of a `char` variable. All pointers defined in CONVEX C refer to the location of a byte in memory. Pointers have the same range of possible values as the `unsigned int` data type. Not all possible unsigned integer values, however, can be used as valid pointers.

While it is not an error for a pointer variable to contain the address of an invalid memory location, it is an error for the program to attempt to access the contents of the address to which such a pointer refers. Also, it is an error for a program to reference an object pointed to by a NULL (0) pointer.

The data representation of a pointer is shown in Figure A-7. Word-aligned pointers have zeros as the two least-significant-bit positions; halfword-aligned pointers have a zero in the least-significant-bit position. Aligned addresses usually result in faster program execution because data with aligned addresses can be cached in high-speed memory by CONVEX hardware. The compiler tries to keep addresses properly aligned.

Figure A–7: Pointer Representation



void Data Type

The void data type does not return any values or perform any operations. This data type specifies the return type of a function that does not return a value and can also be used to discard a return value. This type has no data representation.

The void * data type is a generic data pointer that can be used to point to any data type. It is frequently used in function declarations when the actual return type or parameter is unknown. For example, the function declaration of malloc is:

```
extern void *malloc(size_t);
```

To obtain a pointer to a char object, the return type must be cast:

```
char *cp;
cp = (char *) malloc(4);
```

The representation of the void * data type is the same as that for a pointer shown in Figure A-7.

union Data Type

The union data type permits different data types to be assigned the same storage location. For example, a 32-bit integer can occupy the same memory location as a 32-bit floating-point number.

One common use of the union data type is to provide access to individual bytes in a variable. For example:

```
union {
    unsigned short word_a;
    struct {
        unsigned char hi_byte;
        unsigned char lo_byte;
    } a;
} reg;
```

High and low bytes can be manipulated as individual quantities. If reg.a.lo_byte is 255, adding one to its value does not affect the high byte. In contrast, adding one to the value of reg.word_a may change the value of both the high and the low bytes.

The representation of the union data type depends on the size of the largest member in the union. If the largest member of a union type is a long int, the union type occupies 4 bytes of storage. The address of each member of the union is the same as the address of the union type identifier.

struct Data Type and Representation

A structure is a collection of heterogenous data items that are grouped together under a single name.

ANSI C allows assigning of one structure to another via the "=" operator. Given the declarations

```
struct employee {
    char name[40];
    int age;
    char sex;
    };
struct employee new_emp = {"john smith", 31, 'm'},
         old_emp;
```

`old_emp` can be assigned the values from `new_emp` with the single assignment statement

```
old_emp = new_emp;
```

rather than the three statements

```
strcpy( &old_emp.name, &new_emp.name );
old_emp.age = new_emp.age;
old_emp.sex = new_emp.sex;
```

ANSI C also supports the use of structures in function calls and returns. Structures are passed to functions by value. The entire structure is pushed onto the stack. If `new_emp` is passed to a function update_emp, 48 bytes are pushed on the stack. The extra three bytes maintain stack alignment for performance reasons.

Functions can also be declared to return structure values. For example,

```
struct employee update_emp( struct employee );
new_emp = update_emp( old_emp );
```

The alignment of members within a structure depends on the data types of the members. That is, an `int` member does not cross a 32-bit aligned boundary. This does not imply that all `int` members are aligned on 32-bit boundaries. For example, two 16-bit `int` members fit in the same 32-bit package.

Boundaries for members within structures are the same as the alignment values for variables on the runtime stack.

Structure Padding

Padding and alignment for members of structures are listed below:

- `char`, `unsigned char`, and `signed char` are aligned on byte boundaries.
- `short`, `unsigned short`, and `signed short` are aligned on even byte boundaries.
- `int`, `unsigned int`, and `signed int` are aligned on 4-byte boundaries.
- `long`, `unsigned long`, and `signed long` are aligned on 4-byte boundaries.
- `long long`, `unsigned long long`, and `signed long long` are aligned on 4-byte boundaries.
- `float` is aligned on 4-byte boundaries.
- `double` and `long double` are aligned on 4-byte boundaries.
- The alignment of arrays is dictated by the alignment of each element in the array.
- Structures and unions are aligned as required by the most restrictive member.
- Bit fields are allocated in blocks of size `int`. They begin at the high-order location of storage. Bit fields that span an `int` boundary are placed in the following `int` storage location. Bit fields of type `short`, `char`, and `long long` are also permitted in the extended and backward-compatible modes.

Array Data Type and Representation

An array is an aggregation of data in which all items are one data type. The items are arranged in contiguous memory locations.

Arrays have between 1 and 14 dimensions. For example

```
int four_dim [4][5][6][7];
```

is a declaration of a four-dimensional array that contains 840 items. Arrays are stored in row-major order. All arrays are zero based; the first element of array A is `A[0]`.

Array Addresses

Like more primitive data types, the address of an individual element in an array is specified using the & operator. For example, the address of the third item in a one-dimensional array named `eigen` is:

```
&eigen[2];
```

Consequently, the base address of the array can be specified as `&eigen[0]`. However, the base address of the array can also be specified as `eigen` (or `&eigen` in ANSI C).

When arrays are passed to a function, it is the array name that specifies the array:

```
int some_array[12];       /* array declaration */
void use_array( int [] );  /* function declaration */

use_array( some_array );   /* pass the array */
```

C does not check boundary values on arrays; you must ensure that you do not make illegal array references in `use_array`.

Pointers to Arrays

As noted in the previous section, the name of the array is a constant pointer to the base address of an array. Unlike a pointer, the address of the array cannot be modified. For example, the statement

```
array_name += 5;
```

is incorrect. The name of an array can never be the recipient of an assignment.

However, modifying a pointer to an array is legal. If a pointer is declared as pointing to the same type of data that is contained in the array, that pointer can point to individual elements in the array, and assignments can be made to that pointer. If the pointer is declared to be a different type from the elements in the array, arithmetic operations with the pointer probably will not produce the desired results.

String Representation

A special case of an array representation is the *string* data type. For example,

```
char filename[] = "main.c";
```

defines an array of type `char` that contains the filename `main.c`. Each byte in the array contains an ASCII character code. The NULL byte is automatically appended to the string. The NULL byte indicates the end of the string; it is expected by many system functions that have parameters that are strings.

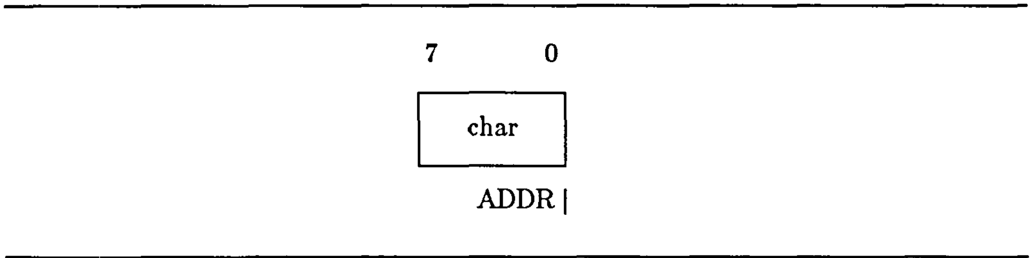
If a string is created by hand, as in:

```
char filename[7];
filename[0] = 'm'; filename[1] = 'a'; filename[2] = 'i';
filename[3] = 'n'; filename[4] = '.'; filename[5] = 'c';
filename[6] = NULL;
```

the NULL character *must* be appended to the string by hand. This character is automatically appended only when arrays are initialized to a string or in some functions such as `strcat` and `strcpy`.

Figure A-8 shows the representation of this data type.

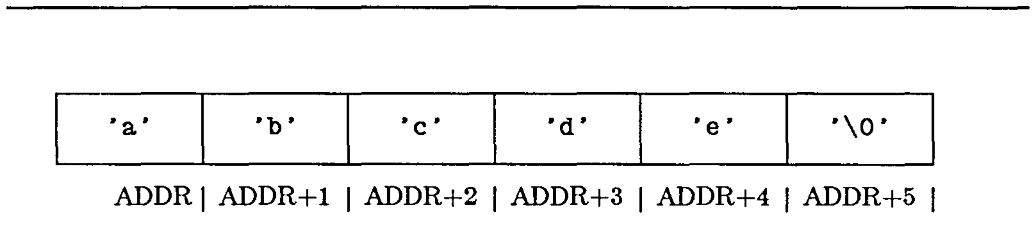
Figure A–8: Character Data Representation



Single-character constants in C are delimited by apostrophes. ASCII codes are specified as `char` variables when you place the one- to three-digit octal number representing the desired character code preceded by a backslash (\ ) character between apostrophes, as in `'\64'`.

Arrays of character data are stored in ascending memory addresses, regardless of 32-bit word boundaries. Figure A-9 shows the representation of the string "abcde" in memory.

Figure A–9: Character String Representation



APPENDIX B
Pragmas

This appendix describes CONVEX C compiler pragmas that affect optimization. Some pragmas listed here provide the compiler with information that it cannot deduce on its own, while others instruct the compiler to override default conditions that control optimization, vectorization, or parallelization.

A pragma for CONVEX C has the form

#pragma _CNX *pragma-name* [*options*], *pragma-name* [*options*] ...

More than one pragma may be specified on each line. The compiler issues a diagnostic message for pragmas that it does not recognize. Pragmas can be used in all the compatibility modes of the compiler.

Certain combinations of pragmas are invalid when used within the same function or loop and cause the compiler to issue a warning. Table B-1 lists invalid pragma combinations.

Table B-1: Restrictions on Pragma Use

Pragma	Cannot be used with
force_parallel	force_parallel_ext, force_vector, no_parallel, prefer_parallel, prefer_parallel_ext, prefer_vector, scalar, select, synch_parallel, unroll, vstrip
force_parallel_ext	force_parallel, no_parallel, prefer_parallel, prefer_parallel_ext, prefer_vector, pstrip, scalar, select, synch_parallel, unroll
force_vector	force_parallel, no_vector, prefer_parallel, prefer_parallel_ext, prefer_vector, pstrip, scalar, select, synch_parallel, unroll
no_parallel	force_parallel, force_parallel_ext, prefer_parallel, prefer_parallel_ext, pstrip, scalar, select, synch_parallel, vstrip
no_side_effects	force_vector, prefer_vector, scalar, select, vstrip
no_vector	force_vector, prefer_vector, scalar, select, vstrip
prefer_parallel	force_parallel, force_parallel_ext, force_vector, no_parallel, prefer_parallel_ext, prefer_vector, scalar, select, synch_parallel, unroll, vstrip
prefer_parallel_ext	force_parallel, force_parallel_ext, force_vector, no_parallel, prefer_parallel, pstrip, scalar, select, synch_parallel, unroll
prefer_vector	force_parallel, force_parallel_ext, force_vector, no_vector, prefer_parallel, pstrip, scalar, select, synch_parallel, unroll
pstrip	force_parallel_ext, force_vector, no_parallel, prefer_parallel_ext, prefer_vector, scalar, synch_parallel, unroll, vstrip
scalar	force_parallel, force_parallel_ext, force_vector, no_parallel, no_vector, prefer_parallel, prefer_parallel_ext, prefer_vector, pstrip, select, synch_parallel, vstrip
select	force_parallel, force_parallel_ext, force_vector, no_parallel, no_vector, prefer_parallel, prefer_parallel_ext, prefer_vector, scalar, unroll
synch_parallel	force_parallel, force_parallel_ext, force_vector, no_parallel, prefer_parallel, prefer_parallel_ext, prefer_vector, pstrip, scalar, unroll, vstrip
unroll	force_parallel, force_parallel_ext, force_vector, prefer_parallel, prefer_parallel_ext, prefer_vector, pstrip, select, synch_parallel, vstrip
vstrip	force_parallel, no_parallel, no_vector, prefer_parallel, pstrip, scalar, synch_parallel, unroll

A pragma associated with a loop affects the loop that immediately follows the pragma and does not affect nested loops. The remaining sections in this appendix describe the pragmas. A pragma's format is shown when it has associated options.

begin_tasks, next_task, end_tasks

A task is a sequence of linear code that can be executed in parallel with other tasks. The begin_tasks pragma identifies a sequence of tasks for independent, parallel execution. The sequence of tasks ends with end_tasks. The next_task pragma precedes each task except the first. The following code shows the use of the tasking pragmas.

```
#pragma _CNX begin_tasks
    <statement>
#pragma _CNX next_task
    <statement>
#pragma _CNX next_task
    <statement>
#pragma _CNX end_tasks
```

You can specify a maximum of 255 tasks between a begin_tasks and an end_tasks pragma.

force_parallel

The force_parallel pragma is effective only if you specify the -O3 compiler option. The pragma tells the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this pragma on a loop whether or not the loop contains calls, but it may not be safe to do so.

Certain actual dependencies, such as from one scalar to another, cause the compiler to ignore the force_parallel pragma. This pragma does not allow the compiler to interchange or distribute loops outside the force_parallel loop for vectorization. To enable those optimizations, use force_parallel_ext.

Caution	This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the parallelized code.

If you use this pragma with scalar or no_parallel, a warning is issued. Also, a warning is issued when you use force_parallel and another parallelizing pragma in the same loop nest. The following code shows how to use force_parallel:

```
#pragma _CNX force_parallel
for( i=0; i<n; i++ )
    sub(a, i);
```

force_parallel_ext

The `force_parallel_ext` pragma is effective only if you specify the `-O3` compiler option. This pragma forces the compiler to parallelize the loop that follows, regardless of apparent dependencies between iterations. You can use this pragma on a loop whether or not the loop contains calls.

If you specify `force_parallel_ext` and `force_vector` for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop. The `force_parallel_ext` pragma allows the compiler to interchange outer loops for vectorization.

> **Caution** This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the parallelized code.

If you use this pragma with `scalar` or `no_recurrence` a warning is issued. Also, an error occurs when you use `force_parallel_ext` and another parallelizing pragma in the same loop nest.

force_vector

The `force_vector` pragma forces the compiler to vectorize the loop that follows, regardless of apparent recurrences. It is possible to use `force_vector` on a loop that the compiler would not fully vectorize without the pragma and get incorrect answers because the pragma causes the compiler to ignore dependencies.

Use this pragma only with fully vectorizable loops. If you specify `force_vector` and `force_parallel_ext` for the same loop, the compiler first vectorizes the loop and then parallelizes the resulting strip-mine loop.

> **Caution** This pragma causes the compiler to ignore any apparent dependencies between iterations. When you use this pragma on a loop, you may not get correct results. Check answers generated by the vectorized code.

If you use `force_vector` with `no_recurrence` or `scalar`, a warning is issued. A warning is also issued when you try to use `force_vector` and another vectorizing pragma in the same loop nest.

max_trips

The max_trips pragma tells the compiler that the loop will execute no more than the specified number of times. The format of this pragma is:

```
#pragma _CNX max_trips(<n>)
```

where the value of <n> is less than or equal to the vector register length of 128. You can use this pragma to prevent the compiler from strip mining the loop. Eliminating strip mining results in more efficient code generation when the maximum trip count is less than or equal to 128.

no_recurrence

The no_recurrence pragma instructs the compiler to disregard any recurrences in a loop. If nothing else impedes vectorization, the compiler vectorizes the loop.

The no_recurrence pragma does not affect recurrences caused by a nested for loop. You can, however, use the pragma on each loop in a nest to give the vectorizer maximum opportunity to improve the nest's performance.

When you use no_recurrence and the compiler finds a recurrence, the compiler breaks the recurrence by removing one or more dependencies of the cycle. In the following code, if j is positive, no recurrence exists.

```
#pragma _CNX no_recurrence
for( i=0; i<n; i++ )
    a[i] = a[i + j];
```

The compiler always accepts a no_recurrence pragma on an apparent recurrence involving an array element; the compiler always ignores a no_recurrence pragma on an apparent recurrence involving a scalar. In the latter case, the compiler knows that a recurrence exists.

Caution	Incorrect results can occur if you mistake a real recurrence for an apparent one. Always test vector results against scalar results to determine whether a recurrence is real or apparent.

For more information about recurrence, refer to the *CONVEX C Optimization Guide.*

no_side_effects

The `no_side_effects` pragma tells the compiler that the specified function does not modify the value of an argument or global variable, perform input or output, or call another subprogram. The pragma applies only to the specified function and not to calls that the function might make.

The format of this pragma is:

```
#pragma _CNX no_side_effects(<func>[,<func>])
```

The argument `<func>` specifies a user-defined function.

This pragma allows the compiler to remove a function call during scalar optimization if the call occurs in an expression assigned to an unused scalar variable. The compiler removes the function call because the function has no side effects. Such optimization opportunities usually arise after the compiler performs other optimizations. These opportunities rarely occur in the original source text.

Place the pragma before the call to the named function but after its declaration. If `<func>` has not been declared, its use in the pragma implies a declaration of `extern int func()`. The following code shows how to use this pragma.

```
int f1(int, int);
#pragma _CNX no_side_effects(f1)
x = y * f1(5, Z) - w;
```

A function call with no side effects is invariant with respect to a loop when:

- its arguments do not vary within the loop and the function call can be moved out of the loop
- it does not modify a nonlocal variable
- it does not perform I/O

no_parallel

The `no_parallel` pragma tells the compiler not to parallelize the loop immediately following the pragma. The pragma does not prevent vectorization of the loop.

If `no_parallel` and `no_vector` both precede the same loop, that loop will run in scalar mode.

no_vector

The `no_vector` pragma tells the compiler not to vectorize the loop immediately following the pragma. This pragma does not prevent parallelization.

If `no_parallel` and `no_vector` both precede the same loop, that loop wil run in scalar mode.

prefer_parallel

The `prefer_parallel` pragma tells the compiler to parallelize the loop immediately following the pragma only if it appears safe. The compiler checks for actual loop-carried dependencies. If the compiler does not find any dependencies, it parallelizes the loop.

This pragma prevents the compiler from interchanging and distributing loops outside the `prefer_parallel` loop for vectorization, whereas `prefer_parallel_ext` does not.

prefer_parallel_ext

The `prefer_parallel_ext` pragma tells the compiler to parallelize the loop immediately following the pragma only if it appears safe. The compiler checks for actual loop-carried dependencies. If the compiler finds no dependencies, it parallelizes the loop.

This pragma allows the compiler to interchange loops outside the `prefer_parallel_ext` loop for vectorization. To vectorize a loop and parallelize the resulting strip-mine loop, use `prefer_parallel_ext` and `prefer_vector` at optimization level `-O3`.

prefer_vector

The `prefer_vector` pragma tells the compiler to vectorize the loop immediately following the pragma only if it appears safe. The compiler checks for actual recurrences. If the compiler finds no recurrences, the compiler tries to interchange the loop so that it is the innermost loop and then it tries to vectorize the interchanged loop.

pstrip

The `pstrip` pragma tells the compiler to strip-mine the parallel loop immediately following the pragma. The compiler strip mines the loop according to the strip-mine length you specify. The format of this pragma is:

```
#pragma _CNX pstrip(<integer_constant>)
```

where `<integer_constant>` specifies the strip-mine length.

The default action of parallel strip mining combines loop iterations into groups of n/(2*ep), where n is the actual loop trip count, and ep is the number of processors (specified with the `-ep` compiler option) when the number of processors is greater than one. If the number of expected processors is one, the number of loop iterations in a group is always one. To override the default, use `pstrip` to specify with `<integer_constant>` the number of iterations to group.

A single thread executes each group. Parallel strip mining occurs only at optimization level `-O3`. If you do not use **`pstrip`**, the compiler selects a default strip-mine length appropriate for the architecture of the machine for which you are compiling.

You cannot use **`pstrip`** with vector loops.

When the number of iterations is small (less than 32), a **`pstrip`** value of one usually gives the best results.

scalar

The **`scalar`** pragma prevents the loop following the pragma from being vectorized or parallelized. This pragma does not prevent other loops from being vectorized or parallelized.

The **`scalar`** pragma is useful when the loop's iteration count is too low for the overhead involved in setting up vectorization, or when you must obtain numerical results identical to those of a scalar loop. You can also use this pragma to prevent the compiler from interchanging or distributing loops. In some cases, the compiler cannot determine the iteration counts of loops and might not choose the best loops to interchange.

The results of a vectorized loop can differ from its scalar version. For example, floating-point sum and product reduction operators can give different answers because the underlying hardware does not process the operands sequentially.

In the following code, the compiler normally interchanges the `i` and `j` loop so that elements of `a`, `b`, and `c` are accessed contiguously. The **`scalar`** pragma ensures that the loop with the greater iteration count is retained as the innermost loop. (The second dimension of `a`, `b`, and `c`, is 1001 to optimize memory accesses. Refer to the *CONVEX C Optimization Guide* for more information on this technique.)

```
float a[2][1001], b[2][1001], c[2][1001];

int sum(int n, int m)
{
    int i,j;

    #pragma _CNX scalar
    for( i=0; i<n; i++ ) /* n = 2 */
        for( j=0; j<m; j++ ) /* m = 1000 */
            a[i][j] = b[i][j] + c[i][j];
}
```

In the following code, neither iteration count is sufficient to warrant vectorizing the loops.

```
float a[2][2], b[2][2], c[2][2];

int sum(int n, int m)
{
    int i,j;

    #pragma _CNX scalar
    for( i=0; i<n; i++ ) /* n = 2 */
        #pragma _CNX scalar
        for( j=0; j<m; j++ ) /* m = 2 */
            a[i][j] = b[i][j] + c[i][j];
}
```

select

The `select` pragma tells the compiler to generate multiple versions of a loop that select runtime code based on specified trip, or iteration, counts. The compiler can generate up to four versions of a loop: scalar, vector, parallel, and parallel-vector. The format of this pragma is:

```
#pragma _CNX select(<vtrip>,<ptrip>,<pvtrip>)
```

The arguments `<vtrip>`, `<ptrip>`, and `<pvtrip>` specify the trip count at which to select vector, parallel, or parallel-vector execution, respectively, for the loop following the pragma. Parallel-vector execution implies that the loop is vectorized and the resulting strip-mine loop is parallelized.

If you omit a trip count by using two adjacent commas, the compiler uses a default value. If you replace a trip count with an asterisk, the compiler does not generate code for the corresponding mode.

If the actual trip count is less than or equal to the smallest specified trip count in the pragma, the loop runs scalar. If the actual trip count is greater than the largest trip count, the loop runs in the mode of the largest trip count. For example, suppose you precede a loop with this statement:

```
#pragma _CNX
select(10,4,200)
```

The loop runs scalar if the actual trip count is 1 to 4, and parallel if the trip count is 5 to 10. The loop runs vector if the trip count is 11 to 200, and parallel-vector if the trip count is greater than 200.

The statement

```
#pragma _CNX select(*,*,*)
```

causes the loop to run in scalar mode.

synch_parallel

The `synch_parallel` pragma is effective only if you specify the `-O3` compiler option. This pragma tells the compiler to generate code that executes the loop that follows in parallel. However, instead of ignoring dependencies, the compiler inserts synchronization code that causes the dependencies to be honored at runtime.

Without specific pragmas, the compiler vectorizes any dependency-free part of the loop; this normally produces superior results. However, if a loop contains much code that is conditionally executed, you might want to parallelize the loop with the `synch_parallel` pragma, particularly if all the dependencies are in seldom executed branches.

On a machine with four processors, the loop in the following code might run faster parallelized and synchronized than if it is partially vectorized and the recurrence placed in a scalar, nonparallel loop.

```
float a[100], b[100], d[100], e[100], f[100];

int calc(int n, int m)
{
    int i;

    #pragma _CNX synch_parallel
    for(i=0; i<32; i++)
        if( a[i] < 0 ){
            a[i] = a[i] + b[i];
            d[i] = e[i] * f[i];
        }
}
```

unroll

The `unroll` pragma is effective only if you specify the `-O2` or `-O3` compiler option. The `unroll` pragma reduces loop overhead by replicating the body of the loop following the pragma. Complete unrolling is performed only on loops that can be vectorized. Partial unrolling is performed on loops that cannot be vectorized.

To be eligible for unrolling, a loop must contain no internal branching and have an iteration count that the compiler can determine. The compiler unrolls a loop completely only if it knows that the iteration count is less than five; otherwise, the compiler partially unrolls the loop. Complete unrolling occurs before vectorization. Partial unrolling occurs after vectorization.

vstrip

The `vstrip` pragma tells the compiler to strip mine the vector loops immediately following the pragma. This pragma is especially useful for automatically parallelized vector loops (loops that are vectorized and run with the outer strip parallel).

The format of this pragma is:

```
#pragma _CNX vstrip(<integer_constant>)
```

where `<integer_constant>` specifies the strip-mine length.

Vector strip mining executes a loop in strips of 128 elements by default, and the parallel outer loop runs iterations of the vector loop in parallel. `vstrip` overrides the compiler default and specifies a different strip-mine length. A shorter strip optimizes the iterations of the strip-mine loop so that the loop can be effectively parallelized. To determine the approximate maximum strip-mine length when the number of expected processors (specified with the `-ep` option) is more than one, the compiler uses the formula:

```
max(min(((n + ep - 1) / ep), 128), 8)
```

where `n` is the actual loop trip count.

The actual strip-mine length is the smaller of the number of iterations remaining to be processed or the maximum length of the strip determined with the formula (either the default or from the pragma).

APPENDIX C
Compiler Directives

Compiler directives are functionally equivalent to pragmas, but use of the compiler-directive syntax is being phased out.

Compiler-directive syntax is maintained for backward compatibility only. No new compiler directives will be added to the compiler. You should use the pragma syntax when inserting directives into your programs.

Compiler-Directive Syntax

A compiler directive of the form

/***$dir** *directive* [, *directive*]*/

is equivalent to

#pragma *directive* [,*directive*]

The directive must begin with the characters **/*$dir**, followed by one or more of the directives in this appendix, followed by */. You may insert one or more spaces or tabs after the initial /* and before the final */. If two or more directives are contained within the /* */, they are separated by commas.

Compiler directives are listed in Table C-1. The description for each of these directives can be found in Appendix B, "Pragmas." Certain combinations of directives and pragmas are invalid when used within the same program unit and will cause the compiler to issue an error message. Table B-1 lists the invalid combinations.

Table C-1: Compiler Directives

begin_tasks	end_tasks	force_parallel
force_vector	max_trips	next_task
no_recurrence	no_side_effects	prefer_parallel
prefer_vector	pstrip	scalar
select	synch_parallel	unroll
vstrip		

APPENDIX D
Cray Intrinsic Functions

Introduction

A subset of Cray's bit intrinsic functions is available with this release of CONVEX C. These functions are available in the extended (default) and backward-compatible mode of the compiler. They are provided to increase the portability of programs from Cray Standard C to CONVEX C.

Even though the functions are implemented using the same name, there are differences between the implementations. These differences include:

intrinsic function mapping
Not all of the Cray bit intrinsic functions can be implemented using intrinsic instructions on a CONVEX computer. Some functions are implemented using function-like macros, while other are implemented using external functions.

bint.h include file
You must declare the functions. The bint.h include file, available in /usr/include, contains the function prototypes for each of the functions. The compiler uses this information to coerce arguments and to increase vectorization.

-lbint command line option
You must include `-lbint` on the `cc` command line so that the compiler can link the bit intrinsic functions with your program. The library that contains the bit intrinsic functions is /usr/lib/libbint.a.

The following functions have been implemented:

- `_count`
- `_dshiftl`
- `_dshiftr`
- `_gbit`
- `_gbits`
- `_ldzero`
- `_leadz`
- `_mask`
- `_maskl`
- `_maskr`
- `_parity`
- `_pbit`
- `_pbits`
- `_popcnt`
- `_poppar`

Function Descriptions

In each of the function descriptions below, the rightmost bit position is zero, so the sixth bit position is the seventh bit from the right.

bint_t _count(bint_t target)
Returns the number of "1" bits contained in `target`.

bint_t _dshiftl(bint_t target, bint_t source, int length);
Replaces the `length` rightmost bits of `target` with the `length` leftmost bits of `source`. Returns the result.

bint_t _dshiftr(bint_t source, bint_t target, int length);
Replaces the `length` leftmost bits of `target` with the `length` rightmost bits of `source`. Returns the result.

bint_t _gbit(bint_t target, int i);
Returns the value in `target` of the bit located in the `i`th bit position from the right.

bint_t _gbits(bint_t target, int length, int i);
Returns the value in `target` consisting of `length` bits beginning with the `i`th bit position from the right. The bits are right justified.

bint_t _ldzero(bint_t target);
bint_t _leadz(bint_t target);
These functions perform the same action. They return the number of leading zeros contained in `target`. Note that the argument is coerced to the `bint_t` data type, a typedef of `long long`. Consequently, the number of zeros may be inflated. Refer to the example at the end of this appendix.

bint_t _mask(bint_t length);
Returns a mask of `length` left-justified "1" bits if 0 ≤ `length` ≤ 63. Otherwise, it returns a mask of 128-`length` right-justified "1" bits.

bint_t _maskl(bint_t length);
Returns a mask of `length` left-justified "1" bits.

bint_t _maskr(bint_t length);
Returns a mask of `length` right-justified "1" bits.

bint_t _parity(bint_t target);
Returns the parity of `target`: zero if it has an even number of "1" bits or one if it has an odd number of "1" bits.

bint_t _pbit(bint_t target, int i, bint_t source);
Returns the value of `target` in which the bit located in the `i`th bit position from the right is replaced by the least significant bit of `source`.

bint_t _pbits(bint_t target, int length, int i, bint_t source);
Returns the value of `target` in which `length` bits beginning with the `i`th bit position from the right are replaced by the value of the `length` least significant bits of `source`.

bint_t _popcnt(bint_t target);
Returns the number of "1" bits contained in `target`.

bint_t _poppar(bint_t target);
Returns the parity of `target`: zero if it has an even number of "1" bits or one if it has an odd number of "1" bits.

Example

The following example computes the population count, the leading zero count, and the parity count of a `char` data type.

```
#include <stdio.h>
#include <bint.h>

main()
{
    unsigned char arg = 0x30;
    bint_t retval;

    retval = _count( arg );
    (void) printf("population count = %lld\n", retval );
    retval = _leadz( arg );
    (void) printf("leading zeros = %lld\n", retval-56 );
    retval = _parity( arg );
    (void) printf("parity = %lld\n", retval );
}
```

Note that the return value of the `_leadz` function is subtracted by 56. This is to account for the extra zeros introduced by the coercion to the `bint_t` data type.

APPENDIX E
Implementation-Defined Features

This chapter states the implementation-defined features of the strict compatibility mode of CONVEX C. Using these features may cause problems when a program is being transferred to or from another computer system. When such a transfer takes place, implementation-defined features of the two compilers must be compared to determine what changes are required in the program.

Implementation-defined features in this chapter are organized according to the ANSI X3J11/90-013 document *American National Standard for Information Systems -- Programming Language C*, Appendix A.6.3, "Implementation-defined Behavior."

All these features pertain only to the strict compatibility mode of CONVEX C, except where noted.

Translation

- A file that contains one or more syntax errors will generate a message in the following form when it is compiled:

  `cc: Error/Warning on line` *line* `of` *file*: *message*

  where

`cc`	indicates a compiler warning or error.
file	is the name of the file that contains the error.
line	is the line on which the error was detected.
message	is the warning or error message that the syntax error generates.

Environment

- The main function has three arguments: `argc`, `argv`, and `envp`. `argc` contains the number of arguments on the program command line, `argv` is an array of string pointers each of which is an argument on the command line, and `envp` is a pointer to an array of strings that constitute the environment of the program. The last array element in `argv` and `envp` is a NULL pointer.

- The interactive devices available to an executable program are in the /dev directory; file names are prefixed by tty, pty, and console.

Identifiers

- Every character in an identifier with *internal linkage* (one that is not visible in another compilation unit) is significant.
- An identifier with *external linkage* (one that is used in another compilation unit) has 41 significant initial characters.
- Characters in an identifier with external linkage are case sensitive.

Characters

- Source and execution character sets are ASCII characters.
- No multibyte character encodings are implemented in this version of CONVEX C.
- There are eight bits in a character in the execution character set.
- Mapping of members of the source character set to members of the execution character set is one to one.
- The value of character constants containing undefined escape sequences will be the same value as the character constant without the back slash. For example, '\c' is the same as 'c'. Also, the value of '~', '@', '$', and any control character within single quotes will be the ASCII value for that character.
- If a character constant contains more than one character, its value is obtained using the following pseudocode:

```
int value = 0;
while( more characters )
    value = ( value << 8 ) + ( value of next character );
```

  Because an integer occupies four bytes of memory, the value of a character constant that has more than four characters will consist of only the last four characters. Character constant representations are shown in Table E-1.

Table E-1: Character Constant Representation

Constant	Representation
'a'	0x61
'abc'	0x616263
'abcdef'	0x63646566

  The type of the result is `signed int`. In the extended and backward-compatible modes, the character constant can hold up to eight characters because `value` can be type `long long int`.

- One locale is available with CONVEX C: `"C"`. Multibyte characters are not implemented.

- A `char` type has the same range of values as a `signed char` type.

Integers

- Integers are represented in two's complement form. Ranges for the integral types are enumerated in Appendix A, "Data Types and Representations."

- Converting an integer to a shorter signed integer is the same as transferring the low order N bits of the source integer, where N is equal to 8 * sizeof(destination). Converting from an unsigned quantity to a signed quantity of the same length does not change the representation; the most significant bit of the unsigned quantity is interpreted as the sign bit. Some of these conversions are shown in Table E-2.

Table E-2: Integer Conversions

int	cast to short
-2147483647	1
65535	-1
unsigned short	**cast to short**
65535	-1
32768	-32768

- Performing a bitwise operation on a signed integer has the same result as a bitwise operation on an unsigned integer of the same value.

- The sign of the remainder on integer division is the same as the sign of the numerator.

- A right shift of a negative-valued signed integral type results in a positive value; sign extension does not occur.

Floating-Point Numbers

- The data representations and ranges of the floating-point numbers are described in Appendix A, "Data Types and Representations."

- When an integral number is converted to a floating-point number that cannot exactly represent the original number, the integral number is rounded to the nearest value that can be represented. When a number of type `float` is converted to a number of type `double`, the mantissa is zero-extended. Such a conversion does not increase the accuracy of the value converted.

- When a floating-point number is converted to a narrower floating-point number, it is rounded to the nearest value that can be represented.

Arrays and Pointers

- The type of integer required to hold the maximum size of an array is `size_t`, defined as an `unsigned int`.

- Pointers and `int` integers can be cast to each other without any changes to the underlying bit pattern.

- The type of integer required to hold the difference between two pointers to elements of the same array is `ptrdiff_t` defined as an `int`.

Registers

The `register` storage class has no effect on register usage except in the presence of `asm` statements.

Structures, Unions, Enumerations, and Bit Fields

- If a member of a union object is accessed after a value has been stored in a different member of the object, the value of the member depends upon the types of the two members.

  For example, given:

  ```
  union {
      short snum;
      int  inum;
  } num;
  ```

  the `short` representation overlays the third and fourth bytes of the `int` representation because members in a union have the same address. Consequently, the first byte of a `short` overlays the third byte of an `int`.

Similarly, if a union object contains two members, a `float` and a `char`, the `char` overlays the exponent and sign bit of the `float`. Refer to Appendix A, "Data Types and Representations," for the representations of data types.

- Padding and alignment for members of structures are as follows:

  - `char`, `unsigned char`, and `signed char` types are aligned on byte boundaries.
  - `short`, `unsigned short`, and `signed short` types are aligned on even byte boundaries.
  - `int`, `unsigned int`, and `signed int` types are aligned on 4-byte boundaries.
  - `long`, `unsigned long`, and `signed long` types are aligned on 4-byte boundaries.
  - `long long`, `unsigned long long`, and `signed long long` types are aligned on 4-byte boundaries.
  - `float` is aligned on 4-byte boundaries.
  - `double` and `long double` types are aligned on 4-byte boundaries.
  - The alignment of arrays is dictated by the alignment of each element in the array.
  - Structures and unions are aligned as required by the most restrictive member.
  - Bit fields are allocated in blocks of size `int`. They begin at the high-order location of storage. Bit fields that span an `int` boundary are placed in the following `int` storage location. The extended compatibility mode permits bit fields to have `char`, `short`, and `long long` data types.

- An `int` bit field is treated as an `unsigned int` bit field.

- The order of allocation of bit fields within an `int` is from the most-significant bit position to the least significant bit position.

- A bit field cannot straddle a word (32-bit) boundary, but can straddle a byte (8-bit) or half-word (16-bit) boundary.

- An enumerated data type is represented as a `signed int`.

Qualifiers

Any object that has a `volatile`-qualified type is accessed when its value is used or modified at optimization level `-no`.

Declarators

The number of declarators that may modify an arithmetic, structure, or union type is at least 12.

Statements

The maximum number of `case` values in a `switch` statement is indefinite. Other limits in the compiler will be reached before any limitation imposed on the number of `case` values is reached.

Preprocessing Directives

- Character constants in conditional inclusion directives are unsigned.
- The method used for locating a file in a `#include` directive depends on its delimiters. A file name in a `#include` directive is delimited by double quotes or angle brackets.

  The search order for a file delimited by double quotes is:

  1. The current directory
  2. The directory specified by the `-I` command line option
  3. The system directory, /usr/include

  The search order for a file delimited by angle brackets is:

  1. The directory specified by the `-I` command line option
  2. The system directory, /usr/include

- As indicated in the previous item, source files in the `#include` directive may be delimited by double quotes.
- Source file character sequences are mapped to the ASCII character set.
- Appendix B, "Pragmas," describes the behavior for each `#pragma` directive encountered by the compiler.
- The system translations of __DATE__ and __TIME__ are always available.

Library Functions

- The `NULL` macro expands to the integer value 0.
- The diagnostic message displayed by the `assert` function is of the form:

  `Assertion failed:` *<integer expression>*`, file` *<name>*`, line` *<number>*

  After the `assert` macro prints the diagnostic message, the `abort` function is executed which results in an IOT trap.

- Information in Table E-3 shows which characters cause the functions `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, and `isupper` to return a true value.

Table E-3: Characters Checked by ctype.h Functions

Function	Returns True Value For
isalnum	any letter or digit
isalpha	any letter
iscntrl	delete character or ASCII character with value < 32
islower	any lower case letter
isprint	ASCII code 32 through 126, inclusive
isupper	any upper case letter

- Table E-4 indicates the values returned by math functions when a domain error occurs. (HUGE_VAL is defined in the math.h header file. It is the largest floating-point value. It may not be representable as a `float`.)

Table E-4: Math Function Return Values

Function	Domain Error Return Value
acos	acos(1)
asin	asin(1)
atan	pi/2
atan2	pi/2
cos	cos(0)
cosh	HUGE_VAL
log	log($\lvert x \rvert$)
log10	log10($\lvert x \rvert$)
pow	pow($\lvert x \rvert$)
sin	sin(0)
sinh	-HUGE_VAL
sqrt	sqrt($\lvert x \rvert$)

- Mathematics functions do not set the integer expression `errno` to the macro ERANGE on underflow errors.
- When the `fmod` function has a second argument of zero, a domain error occurs and a zero is returned by the function.
- Signals that can be used by the `signal` function are listed in Table E-5.
- Table E-5 also lists semantics for each signal recognized by the `signal` function.

Table E-5: Signals and Semantics

Signal	Semantics
SIGABRT	abort
SIGALRM	alarm clock
SIGBUS	bus error
SIGCHLD	to parent on child stop or exit
SIGCONT	continue a stopped process
SIGEMT	EMT instruction and sigabrt
SIGFPE	floating point exception
SIGHUP	process hangup
SIGILL	illegal instruction
SIGINT	interrupt
SIGIO	I/O possible signal
SIGIOT	IOT instruction
SIGKILL	kill (cannot be caught or ignored)
SIGLOST	resource lost
SIGPIPE	write on a pipe with no process to read it
SIGPOLL	I/O possible signal
SIGPROF	profiling time alarm
SIGQUIT	quit
SIGSEGV	segmentation violation
SIGSTOP	sendable stop not from tty
SIGSYS	bad argument to system call
SIGTERM	software termination signal from kill
SIGTRAP	trace trap
SIGTSTP	stop signal from tty
SIGTTIN	background read attempted from control terminal
SIGTTOU	background write attempted to control terminal
SIGURG	urgent condition on I/O channel
SIGUSR1	user-defined signal 1
SIGUSR2	user-defined signal 2
SIGVTALRM	virtual time alarm
SIGWINCH	window changed
SIGXCPU	cpu time limit exceeded
SIGXFSZ	file size limit exceeded

- The default action for each of the signals is listed in Table E-6.

Table E-6: Default Actions for Signals

Signal	Default Action
SIGABRT	terminate program and dump core image
SIGALRM	terminate program
SIGBUS	terminate program and dump core image
SIGCHLD	discard signal
SIGCONT	discard signal
SIGEMT	terminate program and dump core image
SIGFPE	terminate program and dump core image
SIGHUP	terminate program
SIGILL	terminate program and dump core image
SIGINT	terminate program
SIGIO	discard signal
SIGIOT	terminate program and dump core image
SIGKILL	terminate program
SIGLOST	terminate program and dump core image
SIGPIPE	terminate program
SIGPOLL	discard signal
SIGPROF	terminate program
SIGQUIT	terminate program and dump core image
SIGSEGV	terminate program and dump core image
SIGSTOP	suspend process
SIGSYS	terminate program and dump core image
SIGTERM	terminate program
SIGTRAP	terminate program and dump core image
SIGTSTP	suspend process
SIGTTIN	suspend process
SIGTTOU	suspend process
SIGURG	discard signal
SIGUSR1	terminate program
SIGUSR2	terminate program
SIGVTALRM	terminate program
SIGWINCH	suspend process
SIGXCPU	terminate program
SIGXFSZ	terminate program

- When a signal occurs, future occurrences of that signal are blocked until you unblock the signal.
- The default handling is not reset if the `SIGILL` signal is received by a handler specified to the signal function.

- The last line of a text stream does not require a terminating newline character.

- Space characters that are written to a text stream immediately before a newline character are not eliminated.

- No null characters are appended to the data written to a binary stream.

- When a file is opened in the append mode, the file position indicator is initially positioned at the end of the file.

- When a write occurs on a text stream, the associated file is not truncated beyond that point.

- CONVEX C supports unbuffered, fully buffered, and line-buffered file input and output. The `stdin`, `stdout`, and `stderr` streams are line buffered.

  Two functions affect buffering of a file: `setbuf` and `setvbuf`. These functions replace the file input and output buffers, that are automatically allocated by an application, with an array. If the second argument of these functions is `NULL`, the input and output is unbuffered.

- It is possible to have zero-length files.

- A file name may consist of 1 to 256 characters. The characters may not include the *slash* or the null character. File names . and .. have special meanings.

- If the `remove` function is executed on a file that is open, you can continue to read and write to that file until the file is explicitly closed. At that time, the file ceases to exist.

- If an attempt is made to rename a file to a file that already exists, the existing destination file is erased.

- The `%p` conversion specification in the `fprintf` function is used to display the address of a pointer. The address displayed is a hexadecimal number that does not have a `0x` prefix.

- When the conversion specification in the `fscanf` function is `%p`, a hexadecimal number is input. This number must not have a `0x` prefix.

- The '-' character in the scanlist of a `%[` conversion specifier has no special meaning. That is, a '-' character in the input is matched regardless of where the '-' appears in the scanlist.

- Values to which the macro `errno` is set by the `fgetpos` or `ftell` function on failure are listed in Table E-7.

Table E-7: errno values of fgetpos and ftell

errno Value	Condition
EBADF	file is not open
EINVAL	the file position is negative
ESPIPE	file name is associated with a pipe or a socket

- Tables E-8 through E-15 list messages generated by the `perror` and `strerror` functions.

Table E-8: Math Error Messages

errno Value	Error Message
EDOM	Argument too large
ERANGE	Result too large

Table E-9: Nonblocking and Interrupt I/O Error Messages

errno Value	Error Message
EALREADY	Operation already in progress
EINPROGRESS	Operation now in progress
EWOULDBLOCK	Operation would block

Table E-10: NFS Error Messages

errno Value	Error Message
EREMOTE	Too many levels of remote in path
ESTALE	Stale NFS file handle

Table E-11: SystemV Record Locking Error Messages

errno Value	Error Message
EDEADLK	Deadlock situation detected/avoided
ENOLCK	No record locks available
ENOSYS	Function not implemented

Table E-12: Error Messages

errno Value	Error Message
EACCES	Permission denied
EAGAIN	Resources temporarily unavailable
EBADF	Bad file number
EBUSY	Mount device busy
ECHILD	No children
EEXIST	File exists
EFAULT	Bad address
EFBIG	File too large
EINTR	Interrupted system call
EINVAL	Invalid argument
EIO	I/O error
EISDIR	Is a directory
EMFILE	Too many open files
EMLINK	Too many links
ENFILE	File table overflow
ENODEV	No such device
ENOENT	No such file or directory
ENOEXEC	Exec format error
ENOMEM	Insufficient free swap space
ENOSPC	No space left on device
ENOTBLK	Block device required
ENOTDIR	Not a directory
ENOTTY	Not a typewriter
ENXIO	No such device or address
EPERM	Not owner
EPIPE	Broken pipe
EROFS	Read-only file system
ESPIPE	Illegal seek
ESRCH	No such process
ETXTBSY	Text file busy
EXDEV	Cross-device link
E2BIG	Arg list too long

Table E-13: Ipc/Network Argument Error Messages

errno Value	Error Message
EADDRINUSE	Address already in use
EADDRNOTAVAIL	Cannot assign requested address
EAFNOSUPPORT	Address family not supported by protocol family
EDESTADDRREQ	Destination address required
EMSGSIZE	Message too long
ENOPROTOOPT	Protocol not available
ENOTSOCK	Socket operation on nonsocket
EOPNOTSUPP	Operation not supported on socket
EPFNOSUPPORT	Protocol family not supported
EPROTONOSUPPORT	Protocol not supported
EPROTOTYPE	Protocol wrong type for socket
ESOCKTNOSUPPORT	Socket type not supported

Table E-14: Ipc/Network Operational Error Messages

errno Value	Error Message
ECONNABORTED	Software caused connection abort
ECONNRESET	Connection reset by peer
EDQUOT	Disk quota exceeded
EHOSTDOWN	Host is down
EHOSTUNREACH	Host is unreachable
EISCONN	Socket is already connected
ELOOP	Too many levels of symbolic links
ENAMETOOLONG	File name too long
ENETDOWN	Network is down
ENETRESET	Network dropped connection on reset
ENETUNREACH	Network is unreachable
ENOBUFS	No buffer space available
ENOTCONN	Socket is not connected
ENOTEMPTY	Directory not empty
EPROCLIM	Too many processes
EREFUSED	Connection refused
ESHUTDOWN	Cannot send after socket shutdown
ETIMEDOUT	Connection timed out
ETOOMANYREFS	Too many references; cannot splice
EUSERS	Too many users

Table E-15: Tape System Error Messages

errno Value	Error Message
ETPBADFORM	Labeled tape not in proper format
ETPNOSUPPORT	Tape feature not supported
ETPTRUNC	Logical tape record would be truncated

- Functions `calloc`, `malloc`, and `realloc` return a unique pointer if the size requested is zero. This pointer is not NULL.

- The `abort` function closes open files and deletes temporary files.

- The `exit` function returns its argument if that argument is nonzero, EXIT_SUCCESS, or EXIT_FAILURE.

- The set of environment names used by the `getenv` function includes:

  ```
  PATH, HOME, TERM, SHELL, TERMCAP, EXINIT, USER,
  PRINTER.
  ```

  Other environment names may also be defined. Similarly, it is possible for an environment to have no environment names. The `setenv` function can be used to change the environment list used by the `getenv`.

- The `system` function uses the same arguments as the `sh` command. When system is executed, the current process waits until the shell has completed, then returns the exit status of the shell.

- The contents of the error message strings returned by the `strerror` function are listed in Tables E-8 through E-15.

APPENDIX F
C Manpages

This appendix contains the C man pages that are not available with other CONVEX documentation.

The previous release of C documentation included *CONVEX C Programmer's Reference.* This document is no longer available. Refer to *ConvexOS Manpages for Programmers* for the man pages that you cannot find in this appendix.

The man pages available in this appendix are:

cc(1)	CONVEX C compiler
cpp(1)	CONVEX C preprocessor
lint(1)	CONVEX C program verifier
intro(3bit)	Introduction to Cray compatible bit manipulation functions.
bint.h(3bit)	Description of bint.h include file.
bitchange(3bit)	Functions that get or set the value of a bit.
bitcount(3bit)	Functions that perform population count, leading zero count, and parity count of bits.
bitmask(3bit)	Functions that create a mask of contiguous "1" bits.
bitmil(3bit)	Function-like macros that emulate the ibits, ibset, ibclr, and mvbits MIL standard bit manipulation functions.
bitshift(3bit)	Functions that shift and rotate bits.

NAME

cc – CONVEX C compiler

SYNOPSIS

cc [*option* ...] *file* ... [*loader_option* ...]

DESCRIPTION

cc is the CONVEX C compiler. It accepts several types of arguments:

NAME

cc – CONVEX C compiler

SYNOPSIS

cc [*option* ...] *file* ... [-link *loader_option* ...]

DESCRIPTION

cc is the CONVEX C compiler. It accepts several types of arguments:

Filenames whose names end with ".c" or ".C" must be C source programs. They are compiled and each object program is placed in a file in the current directory with the same root name as the source filename and a suffix of ".o". For example, compiling /x/y/z.c will place the object code in ./z.o.

Filenames which end with ".s" must be assembly source programs and are assembled, producing a ".o" file.

Other files can be object files or libraries; they are passed to *ld* to be linked together with the objects from all compilations and assemblies. If a single source file is compiled and loaded by a *cc* command, the ".o" file is deleted.

COMPATIBILITY

The CONVEX C compiler operates in one of four compatibility modes.

–ext This mode provides an extended implementation of ANSI C and an ANSI C and POSIX compatible library system. This is the default.

–std This option causes the compiler to operate as an ANSI C conforming compiler. Certain extensions, notably the long long type, are not available in this mode. When linking occurs in this mode an ANSI C and POSIX P1003.1 conforming library system is used; many of the extensions provided by the default library system are not available.

–str This option causes the compiler to operate as an ANSI C conforming compiler and to attempt to detect features of the program which cause it not to conform to ANSI Standard C. When linking occurs in this mode, only the functions defined by ANSI Standard C are available.

−pcc Forces language and library interpretation based on the original Kernighan and Ritchie definition, the Common C Compiler, and traditional Unix systems. This mode is compatible with CONVEX C V3.0 and other C compilers previously shipped by CONVEX.

These options are interpreted before all others, regardless of their position on the command line.

OPTIMIZATION OPTIONS

There are two versions of the C compiler available: one that is capable of only scalar optimizations and another that can perform scalar, vector, and parallel optimizations. The former is available on all CONVEX computer systems, while the latter is an optional product available only to licensed sites. You can determine which product you have by attempting to compile a program with the *-O2* option. An error message will be generated if you do not have the fully optimizing version.

−alias standard

−alias cautious

−alias worst

The -alias option affects the assumptions the compiler makes regarding overlapping of variables, particularly when referenced by a pointer. *standard* causes the compiler to assume the program obeys the ANSI C standard. *worst* causes the compiler to do worst case alias analysis; it assumes that a pointer may reference any piece of memory other than an *auto* variable whose address has never been taken. *cautious* causes the compiler to behave as it does for *standard* except when casts between pointer types are seen. In that case it behaves like *worst.*

Incorrect code may be generated if the implied conditions do not hold.

When the *-pcc* flag is provided, *worst* is the default, otherwise, the default is *cautious.*

−alias array_args

Assume that formal array parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). **Incorrect code will be generated if this assumption is not true.** This conflicts with the language definition, but may allow greater optimization, particularly vectorization, to occur.

This option may not be used if the formal parameter itself is assigned by the function (for example, formalParameter = &x[10]); assignments may be made to the elements of the formal (for example, formalParameter[10] = x[10]).

–alias ptr_args
Assume that the variables identified by formal pointer parameters do not overlap each other or any external variable that is used in the function (unless all uses are read-only). **Incorrect code will be generated if this assumption is not true**. This conflicts with the language definition, but may allow greater optimization, particularly vectorization, to occur.

This option may not be used if the formal parameter itself is assigned by the function (for example, formalParameter = &x[10]); assignments may be made to the elements of the formal (for example, formalParameter[10] = x[10]).

–ds Causes the compiler to perform dynamic selection on loops selected by the compiler on the basis of profitability. Multiple copies of the loop running sequential, vector or parallel modes are generated; the optimal version gets executed based on the trip count of the loop. This option is only effective at optimization level *-O2* and *-O3*.

–ep n Specifies the expected number of processors that will execute the program. Optimizes single-program performance at the expense of overall system throughput. n must be in the range 1 through 4.

–no No machine-independent optimization is performed; this is the default but can be overridden by the *-O* options.

–O*n* Performs machine-independent optimizations, level *n*, where:
n=0 selects basic block scalar optimization;
n=1 selects *-O0* plus function-wide scalar optimization;
n=2 selects *-O1* plus vectorization;
n=3 selects *-O2* plus parallelization.
If this option is not specified, the compiler performs no machine independent optimization (*-no*).

–rl This option is equivalent to *-ds -ur*. It is only effective with options *-O2* and *-O3*.

–uo Performs potentially unsafe optimizations, i.e., may move the evaluation of common subexpressions and/or invariant code from within conditionally executed code. The code is then executed unconditionally.

–ur Causes the compiler to perform loop unrolling on loops selected by the compiler on the basis of profitability. This option is only effective at optimization levels *-O2* and *-O3*.

–va A synonym for *-alias array_args*.

CODE GENERATION OPTIONS

–asm This flag is silently ignored; it is no longer required.

–c Do not run the linker. The object module generated from *x.c* or *x.s* is left in *x.o*.

–extern distinct

–extern same
The *-extern* option may be used to control the interpretation of a program in which different files declare an uninitialized variable (for example, "`int i;`") at file scope. When *same* is used both files refer to the same variable. This is the default in all modes, is traditional on BSD Unix Systems and is a conforming extension to Standard C. When *distinct* is used the files refer to distinct variables; this results in a multiply defined symbol being detected by the linker. This is the definition provided by the ANSI C standard.

–fd A synonym for *-float sp_ops*, described below.

–fi Translate floating-point constants to IEEE format and perform floating-point operations in IEEE mode. This option requires that the machine be equipped with IEEE floating-point hardware. If no floating-point format is specified, the site default is used. Arithmetic performed under this option does not conform to the IEEE standard.

–float sp_ops

–float dp_ops

–float sp_const

–float dp_const
These options control the floating-point system used by the compiler. *sp_ops* and *dp_ops* control the precision of the operations performed on float operands. When *sp_ops* is specified, 32 bit operations are performed. When *dp_ops* is specified, the float operands are converted to double and 64 bit operations are performed. *sp_ops* generally causes programs using float variables to run faster but some accuracy may be lost.

sp_const and *dp_const* control the representation of unsuffixed floating-point constants (which are normally represented in double precision). *sp_const* causes these constants to be represented in single precision. Some loss of accuracy may result. *dp_const* explicitly invokes the default.

Use of *sp_ops* is more effective when combined with *sp_const* since expressions like $x == 0.0$ are performed in double precision when only *sp_ops* is specified. *sp_ops* is the default in all but *-pcc* mode.

dp_const is the default in all modes.

–fn Translate floating-point constants to native CONVEX format and perform floating-point operations in native mode. If no floating-point format is specified, the site default is used.

–parens ignore

–parens explicit

–parens implicit

Toggle the manner in which parentheses and the ANSI C grammar are used to determine the order of evaluation of expressions.

The value *ignore* gives the compiler freedom to re-order expressions; even explicit parentheses are ignored. This is the default in *-pcc* and *-ext* modes and is the traditional method used in C.

The value *explicit* tells the compiler to honor explicit parentheses but ignore ordering implied by the grammar and associativity rules. This is similar to the rules used by Fortran and many other languages.

The value *implicit* tells the compiler that explicit parentheses and ordering implied by the grammar and associativity rules should be honored. This means that a+b+c must be evaluated in the order implied by (a+b)+c. This is the default in *-std* and *-str* modes.

These rules apply only to floating-point expressions; operations on other types are always subject to re-ordering.

–re Specifies that the compiler generate re-entrant code by generating both a scalar and a parallel version of parallel loops. The default, at optimization level *-O3*, is to generate non-reentrant code for functions with parallel regions. This option has no effect on functions without parallel code; it should be used to compile a function for which the compiler generates parallel code, if you wish to call that function from a parallel region. It is always **safe** to use *-re*; the text segment will be unnecessarily large if

-re is used when it is not required.

–S Generates symbolic assembly code for each program unit in a source file. Assembler output for a source file *x.c* is put on file *x.s*; no object file is generated. The default is to write only an object file.

–sc Instructs the compiler to check the program for compilation errors. No optimization, vectorization, or code generation is performed.

–string read_write
String constants are stored in the data segment and may be modified. This is the default when *-pcc* is specified.

–string read_only
String constants are stored in the text segment and may not be modified. This option increases the sharability of a program when multiple copies of it are running at once. This is the default if *-pcc* is not specified.

–tm x Specifies the target machine architecture. *x* may be c1, C1, c2, or C2; the default value is the type of machine the compiler is running on. If c2 or C2 is used, the resulting code will not run on a C1. If c1 or C1 is used, the resulting code will run on either a C1 or a C2 but may run less quickly on a C2 than it would if compiled with *-tm* C2. When invoking the linker machine dependent versions of some libraries are used.

DEBUGGING AND PROFILING OPTIONS

–cxdb Produces additional information for use by CXdb, the CONVEX visual debugger. No additional information is generated if the source file contains *asm* statements or if the **-S** or **-pb** command line options are used. This flag can be used with all levels of optimization, but unless the **-no** option (NO optimization) is specified, there may be source statements for which no debugging information is generated for CXdb. When this option is included on the command line, a subdirectory called .CXdb is created that contains auxiliary information. Refer to *CONVEX CXdb User's Guide* for more information. CXdb is an optional product.

–db Produces additional information for use by the symbolic debugger, *csd*, and the post-mortem dump utility (*pmd*). It also passes the **-lg** flag to the loader. This flag can be used with all levels of optimization, but unless the **-no** option (NO optimization) is specified, there may be source statements for which no debugging information is generated for *csd*. Information about variables declared in the inner blocks of functions is produced only at level *-no*.

–p Causes the compiler to produce code that counts the number of times each routine is called. If loading takes place, replaces the standard startup routine by one that automatically calls *monitor*(3) at the start and arranges to write out a "mon.out" file at normal termination of execution of the object program. A profiled library is searched, instead of the standard C library. An execution profile can then be generated by use of *prof*(1) (optional product).

–pa Produces counting code for routine-level and loop-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

–pab Produces counting code for block-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

–par Includes instrumentation for routine-level profiles using the CXpa utility. Refer to the *CONVEX CXpa User's Guide* for more information. CXpa is an optional product.

–pb Causes the compiler to produce statement-level counting code that produces an execution profile named *bmon.out* at normal termination. Listings of source-level execution counts can then be obtained using *bprof*(1). *bprof* is an optional product.

–pg Causes the compiler to produce counting code in the manner of **-p**, but invokes a runtime recording mechanism that keeps more extensive statistics and produces a *gmon.out* file at normal termination. An execution profile can then be generated by use of *gprof*(1) (optional product).

COMPILER OUTPUT OPTIONS

–d *name*

–d *name=e*

–d *name=w*

The *-d* option provides low level control of diagnostic output. The form *-d name* suppresses the named message. The form *-d name=w* causes the named message to be reported as a warning. The form *-d name=e* causes the named message to be reported as an error. The message names are documented in the following section.

–nv Suppresses all vectorization summary messages (synonymous with *-or none*).

–nw Suppresses all warning diagnostic messages.

–**or** *table*

Specifies the contents of the optimization report to be produced; either the loop table, the array table, or both can be displayed. The value for *table* can be *none, all, array* or *loop.* If this option is not specified, only the loop table is displayed.

DIAGNOSTIC OPTIONS

The following names can be used to control specific messages. The CONVEX C documentation on C compiler error messages provides examples for each of these diagnostic options.

arg_ptr_qual	Detects actual function parameters that do not have the same type qualifiers as those declared in the function prototype.
arg_ptr_ref	Detects actual function parameters that are not the same pointer type as those declared in the function prototype.
assign_in_condition	Detects assignment expressions in locations where conditional expressions are expected, such as *for* statements and *if* statements.
class_ignored	Indicates that an explicit storage class is used to declare a *struct* tag, *union* tag, *enum* tag, or *enum* member.
const_not_init	Detects uninitialized constant variables.
division_by_zero	Detects compile time division by zero.
dollar_names	Detects identifiers embedded with the $ character.
escape_range_sequence	Detects when an escape sequence in an integer constant is greater than 0xff.
float_suffix	Detects when the floating-point suffixes, f, F, l, or L, are used. This option has no effect in the strict and conforming compatibility modes.
function_parameter	Reports when a function is used as a function parameter; only function pointers can be used as a function parameter.

hidden_arg	Indicates that the parameter of a function is hidden by an identifier with the same name declared in the outermost block of that function. This diagnostic affects only the backward-compatible mode.
hidden_extern	Indicates that a declaration using *extern* inside a function definition causes an identifier not in the block to be obscured. This diagnostic affects only the backward-compatible mode.
hides_outer	Indicates that an identifier prevents access to an identifier of the same name in an enclosing block.
implicit_decl	Detects when an implicit declaration is used because a function has not been declared previously.
integer_overflow	Detects when an integer constant larger than 64 bits is used.
long_long_suffix	Detects when the integer literal suffixes `LL` or `ll` are used. These suffixes are permitted only in the extended and backward-compatible modes because the *long long* data type is not available in the strict and conforming compatibility modes.
negative_to_uns	Indicates when a negative constant is assigned to an unsigned type.
no_arg_type	Checks for function arguments declared without a data type.
no_external_declaration	Detects when no declarations are accessible to other compilation units.
non_int_bit_field	Check for bit fields that have type other than *int* or *unsigned int*.
nothing_declared	Detects empty declarations such as *int;*.
null_effect_expression	Detects intrinsic math function calls that have no effect.
pointer_alignment_efficiency	Checks for operations that can result in inefficient memory alignment.

pp_argcount	Indicates that the number of arguments in the actual argument list of a macro does not match the number of arguments in the formal argument list.
pp_argsended	Indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis.
pp_badstr	Indicates incorrect use of #, the stringizing operator that is used in macro definitions.
pp_badtp	Indicates that the result of combining the left and right operands of the ## macro definition operator is illegal.
pp_extra	Indicates that a preprocessor directive has extra arguments. One common example is when the *#endif* directive is followed by text.
pp_idexpected	Indicates that a *#ifdef* or *#ifndef* does not have a legal identifier as an argument, or a formal argument of a function-like macro is not a legal identifier.
pp_line_range	Indicates the argument for the *#line* preprocessor directive does not have a value between 1 and 32767, inclusive.
pp_macro_arg	Indicates that two or more formal macro arguments have the same name.
pp_macro_redefinition	Indicates that the replacement list for a function-like macro is not the same as its original definition. The preprocessor uses the most recent replacement list.
pp_macro_redefinition_cmdl	Indicates that the replacement list for a function-like macro is not the same as its original definition on the command line. The preprocessor uses the most recent replacement list.
pp_malformed_directive	Indicates that the proper syntax for a preprocessor directive was not used.
pp_old_dir	Indicates that the comment style syntax for a compiler directive (pragma) was used.
pp_parse	Indicates that a `#if` directive has an invalid integer expression.

pp_undef	Indicates that the argument of the `#undef` directive is not permitted.
pp_undef_cmdl	Indicates that you attempted to undefined a macro that cannot be undefined on the command line.
pp_unrecognized_directive	Detects a preprocessor directive that the preprocessor does not recognize.
pp_unrecognized_pragma	Detects a pragma that the compiler does not recognize.
shift_too_large	Checks for constant right operands of shift operands that are too large.
short_cvt_truncates	Indicates that an integral type larger than a `short` integer is being assigned to a `short` integer.
skip_to_char	This diagnostic indicates that the compiler is skipping to a character to recover from an error.
skip_to_eof	This diagnostic indicates that the compiler must skip to the end of a file to recover from an error.
strict_syntax	Detects the lack of a semicolon after the last member in a `struct` or `union` declaration or the presence of a comma after the last enumeration constant in an `enum` declaration list.
unsigned_suffix	Detects use of the integer literal suffixes *U* or *u*. This option is not effective with the *=e* setting.

MISCELLANEOUS OPTIONS

–B*dir* Finds substitute compiler (*cocc, cpp,* and *errmsgc*) in the directory *dir.* If *dir* is not specified the standard backup version in the directory */usr/lib/oldcc* is used instead. For example, the command `cc -B/usr/new` would invoke */usr/new/cocc* instead of the default */usr/lib/cc/cocc.*

–o*name*

Specifies that *name* is the name of the executable file produced by *ld.* If this option is not specified, the default name is *a.out.* If **-c** was specified and there is only one file to compile or assemble, *name* is the name of the object module produced.

–tl *time*

Sets the maximum CPU time limit on compilations to *time* minutes. If the CPU time exceeds the allotted time, the compilation is aborted.

-vn Identifies compiler version. Outputs the version numbers of *cc*, *cpp*, *cocc*, and *errmsgc* to *stderr*.

UNIX COMPATIBILITY OPTIONS

The following options are provided for compatibility with compilers of other vendors.

-g A synonym for *-db*.

-n Ignored with a warning.

-O A synonym for *-O1*.

-OL A synonym for *-O1*.

-V A synonym for *-vn*.

-w A synonym for *-nw*

PREPROCESSOR OPTIONS

The CONVEX C compiler invokes the C macro preprocessor *cpp* on each file and compiles the resulting text. The following options affect the preprocessing phase of translation.

-C

-D*name*

-D*name*=*def*

-I*dir*

-P

-U*name*

These options are passed directly to *cpp(1)* without interpretation by *cc*. See *cpp(1)* for further details.

-E Run *cpp* on the named C source files and send the result to the standard output stream (*stdout*). The source is not checked for errors and no object, assembly or executable code is produced.

-k Run *cpp* on the named C source files and generate *make* dependency descriptions on the standard output stream (*stdout*).

If the compiler option *-pcc* is present it is passed to each invocation of the preprocessor.

PREDEFINED SYMBOLS

The following macros are predefined when *cc* invokes *cpp*:

_CONVEX_SOURCE

_POSIX_SOURCE

The symbols _CONVEX_SOURCE and _POSIX_SOURCE are predefined in extended mode. In *-std* mode the user will generally want to define the symbol _POSIX_SOURCE to make the POSIX symbols available in the include files.

_CONVEX_FLOAT_

_IEEE_FLOAT_

The symbol _CONVEX_FLOAT_ is defined when the compiler is operating in native float point mode, _IEEE_FLOAT_ is defined in IEEE mode. See the description of the options *-fi* and *-fn* above.

__STDC__

The symbol __STDC__ is defined as the decimal constant 1 when either the *-std* or *-str* flags are specified. The definition of this symbol indicates that the compiler conforms to the ANSI C Standard. This definition may not be removed by the *-U* option or *#undef.*

__stdc__

The symbol __stdc__ is defined as the decimal constant 1 in all modes except the *-pcc* mode. It indicates that the compiler is an ANSI style compiler, but not that it is conforming. The conforming modes define both __stdc__ and __STDC__.

__NO_INLINE

The symbol __NO_INLINE is defined in *-str* and *-std* modes to suppress recognition of macros which define certain library functions. This is necessary to get conforming implementations of some functions which would otherwise not set *errno* in a standard conforming manner. Refer to *intro*(3m) for more details.

__convexc__

This symbol is always defined. It should be used to identify the compiler. Other symbols defined by the preprocessor (see *cpp(1)*) identify the machine and operating system.

convexvc

This symbol is defined in *-pcc* mode. It's use is obsolescent; the symbol __convexc__ should be used instead.

The preprocessor defines a number of symbols as well; see *cpp(1)* for a discussion of these symbols.

Other names beginning with "__" or _[A-Z] may be predefined by *cc*. Such names are reserved to CONVEX; their usage or availability may change in subsequent releases. Applications should not depend on the *presence or absence* of such names except as defined above.

LOADER USAGE

A number of compiler options impact the manner in which *cc* invokes the loader. Options for the loader may also be specified directly by the user.

CONVEX recommends always using the appropriate compiler to invoke the loader rather than invoking it directly. This will insulate the program from changes in library structure when new compiler releases are installed.

The compiler always passes the flags *-X*, *-NL*, and *-L/usr/lib* to the loader. The compiler flags *-fi* and *-fn* are passed to the loader (in addition to the effects they have on the compiler). The compiler option *-o* causes a similar *-o* option to be passed to *ld*. *-Eposix* is passed to the loader except when *-pcc* is given, in which case *-Enoposix* is passed.

The compiler compatibility mode controls the libraries searched by the loader; this is normally accomplished by passing one or more *-l* options to the loader.

The following options, when present on the *cc* command line are passed to *ld*(1):

-A -D -E -F -L -M -T -X -d -e -l -m -r -s -t -u -x -y

See the *CONVEX Loader User's Guide* for their meaning and use. The *-l* option must be specified after all object files on the *cc* command line to be effective. Loader options which require a value (for example, *-L* and *-E*) must be written with no spaces between the flag and value (e.,g., -L/mydir) when passed through *cc*.

The *-link* option causes the following argument to be passed to *ld*. If the following argument does not start with - or starts with *-l* the argument is added to the loader's file list, otherwise it is added to the loader's flag list. For example, *-link -v3.2.8.5 -link -link* passes the flag *-v3.2.8.5* to the loader causing it to set the version number of the executable, and passes *-link* in the file list causing the loader to search the library libink.a.

OBJECT FILE COMPATIBILITY

When invoked without the *-pcc* flag CONVEX C V4.1 generates object files which are not compatible with object files created by previous compilers.

When invoked with the *-pcc* flag, object files created by previous C compilers may be linked with those created by V4.1; the *-pcc* flag must be used when linking the objects and the cc V4.1 compiler must be used to perform the link step.

Object files created by any mode of CONVEX C V4.1 may be mixed with object files created by any other mode of CONVEX C V4.1.

Object files created by CONVEX C V4.1 should not be linked using other C compilers.

ENVIRONMENT VARIABLES

The CONVEX C compiler prepends the value of the environment variable **CCOPTIONS** (if it is set) to each command line so that options need not be specified every time *cc* is invoked. For example,

setenv CCOPTIONS '-O2'

causes all compilations to be done at *-O2* and

setenv CCOPTIONS '-pcc'

causes all compilations to be done in backwards compatible mode.

The preprocessor *cpp* has a similar variable *CPPOPTIONS* which can be used to affect its behavior when it is invoked directly. However, *CPPOPTIONS* does not have any effect when the compiler invokes *cpp* (the compiler removes the variable before invoking *cpp*).

FILES

file.c	C language input file
file.o	object code output file
a.out	executable output file
/tmp/cocc*	temporary file
/lib/cpp	preprocessor
/usr/lib/cc/cocc	compiler
/usr/lib/cc/errmsgc	compiler error message text
/bin/cc	compiler control program
/usr/lib/crt/crt0.o	runtime start off
/usr/lib/crt/mcrt0.o	startup routine for prof profiling
/usr/lib/crt/gcrt0.o	startup routine for gprof profiling
/usr/lib/crt/bcrt0.o	startup routine for bprof profiling
/usr/lib/libbint.a	Cray-compatible bit intrinsics library
/usr/lib/libc_old.a	backward mode library
/usr/lib/libc.a	extended mode library
/usr/lib/libp1.a	standard mode library
/usr/lib/libansic.a	strictly conforming mode library
/usr/lib/libC1.a	C1 machine dependent library

/usr/lib/libC2.a	C2 machine dependent library
/usr/lib/libC1_old.a	backward compatible C1 machine dependent library
/usr/lib/libC2_old.a	backward compatible C2 machine dependent library
/usr/include	standard directory for "#include" files
/lib/bscan	optional bprof scanner
mon.out	file produced for analysis by *prof*(1)
gmon.out	file produced for analysis by *gprof*(1)
bmon.out	file produced for analysis by *bprof*(1)

Each library has a profiled version whose name is formed by by inserting _p before the .a.

BUGS

See the CONVEX C release notes in /usr/doc for a description of known bugs causing wrong answers.

The compiler will generate incorrect code for a program containing a function call such as f(g(),g()) if g() returns a struct.

SEE ALSO

adb(1), as(1), cpp(1), csd(1)(optional product), cxdb(1)(optional product), gprof(1)(optional product), ld(1), prof(1)(optional product), bprof(1)(optional product), monitor(3), a.out(5)

CONVEX C Guide
CONVEX C Optimization Guide
CONVEX C Quick Reference
CONVEX Loader User's Guide
B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition,* Prentice-Hall, 1988
B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall, 1978
B. W. Kernighan, *Programming in C—a tutorial*

DIAGNOSTICS

The *CONVEX C Guide* provides detailed information regarding error messages produced by the compiler.

A successful compilation is indicated by returning status 0.

Occasional messages may be produced by the assembler or loader.

NAME

cpp – C preprocessor

SYNOPSIS

/lib/cpp [*option*] ... [*infile* [*outfile*]]

DESCRIPTION

cpp is a macro preprocessor similar to the preprocessor used by the CONVEX C compiler *cc*(1). *cpp* is an ANSI C compatible preprocessor that can also be made compatible with previous versions of *cpp* with the *-pcc* option (described below).

COMMAND LINE OPTIONS

If no files are specified *cpp* reads lines from *stdin*, processes them, and writes them to *stdout*. *infile* and *outfile* may be specified to change this. There is no way to specify an output file without specifying an input file.

The following command line options are interpreted by *cpp*:

–C Retains comments in the output. C style comments are usually removed.

–D*name*
–D*name*=*def*
Defines the *name* (as if by *#define*) as *def*, or 1 if *def* is omitted.

–I*dir* Adds *dir* to the search list for *#include* files.

–k Runs the preprocessor on the named C source file and generates a dependency description for make(1); results are printed out on the standard output stream. This option cannot process input from *stdin*.

–P Do not insert control lines into the output.

–pcc Behave compatibly with earlier preprocessors that were not ANSI C conforming.

–U*name*
Remove any initial definition of *name*. Names predefined by the preprocessor are discussed below.

–stdc Adhere strictly to ANSI C requirements.

–vn Identifies version. Outputs the version number of *cpp* to *stderr*.

Lines in the input file that have a "#" character as the first nonblank character are interpreted as directives by *cpp*. The effect of a directive lasts until the end of the source or until it is countered by another directive.

ENVIRONMENT VARIABLES

The CONVEX C preprocessor *cpp* prepends the value of the environment variable
CPPOPTIONS (if it is set) to each command line so that options need not be specified every time *cpp* is invoked.

For example,

setenv CPPOPTIONS '-pcc'

causes all preprocessing to occur in the backward-compatible mode.

The CONVEX C compiler and *lint* remove the value of CPPOPTIONS before invoking the preprocessor to avoid conflict. CPPOPTIONS is useful when invoking *cpp* as a general purpose preprocessor.

PREDEFINED NAMES

The names that are predefined by the preprocessor depend on the options used.

The identifiers "__convex__" and "__unix__" are always predefined as 1.

The identifiers "__LINE__" and "__FILE__" are always predefined; they return the current line number and file name respectively. These definitions may not be removed with #undef or *-U*.

In ANSI conforming mode the identifiers "__DATE__" and "__TIME__" are predefined; they return the current date and time respectively. These definitions may not be removed with #undef or *-U*.

When invoked with the *-pcc* flag the names "convex" and "unix" are predefined. Applications should use __convex__ and __unix__ instead of these names.

When invoked by the CONVEX C compiler other names may be predefined. See *cc*(1) for details.

Other names beginning with "__" may be predefined by *cpp*. Such names are reserved to CONVEX; their usage or availability may change in subsequent releases. Applications should not depend on the *presence or absence* of names beginning with "__" (except those defined above).

TOKEN REPLACEMENT

A control line of the form:

#define *identifier replacement-string*

is a simple macro definition. It causes *cpp* to replace subsequent instances of the identifier with the replacement-string. A line of the form:

#define *identifier(argument, ...) replacement-string*

is a macro definition with arguments. Subsequent instances of the identifier are replaced by the replacement-string in the definition. Each occurrence of an argument mentioned in the formal parameter list of the definition is replaced by the corresponding actual argument from the call.

The ## token-pasting operator, which may appear only in the replacement-string of #define directive, causes catenation of its operands.

Thus

```
#define x a ## b
```

is equivalent to

```
#define x ab
```

The token-pasting operator is not available when *-pcc* is specified.

The # stringizing operator may only appear immediately before a formal parameter name in the replacement-string of a #define directive. When expansion of the macro occurs the actual parameter becomes a string constant. Thus

```
#define f(x)      #x[0]
f(hello)
```

produces

```
"hello"[0]
```

This operator is not available when *-pcc* is specified.

A long macro definition may be continued on another line by adding "\" at the end of the line to be continued.

The replacement string is rescanned for more defined identifiers.

A control line of the form:

#undef *identifier*

causes the identifier's macro definition to be deleted.

FILE INCLUSION

A control line of the form:

#include "*filename*"

or

#include <*filename*>

causes the replacement of that line by the entire contents of *filename*. Files included by a #include statement may contain further *#include* statements; include files may be nested.

The directory search order for *#include* files is (1) the directory of the file that contains the request unless the form **#include** <*filename*> was used (2) the directories specified by *-I*, and (3) the standard directory (*/usr/include*).

CONDITIONAL INSERTION

A control line of the form:

#if *constant-expression*

checks whether the constant expression evaluates to a nonzero quantity. A constant expression is any legitimate C expression that evaluates to a constant. See

"The C Programming Language" for more information. A control line of the form:

#ifdef *identifier*

checks whether the identifier is currently defined in *cpp*. A control line of the form:

#ifndef *identifier*

checks whether the identifier is currently undefined in *cpp*.

Each form of the if statement may be followed by an arbitrary number of lines of text, optionally followed by

#elif *constant-expression*

and more lines of text. Multiple *# elif* constructs may be used.

After all the #elif lines a

#else

line may appear. The conditional input section is terminated by a

#endif

line.

The text is included or omitted according to the truth of the expression.

The expression on a #if or #elif line is subject to macro expansion.

These constructs may be nested.

LINE CONTROL

For other preprocessors that generate C programs, a line of the form:

#line *integer-constant [string-constant]*

causes the C compiler to believe, for purposes of error diagnostics, that the number of the next source line is given by the integer-constant and the current input file is given by the string-constant. If the string-constant is absent, the current filename does not change.

COMMENTS

C comments are replaced by a blank unless the *-C* option or the *-pcc* option is specified.

When the *-C* option is specified comments are retained in the output.

When the *-pcc* option is specified (without *-C*) comments are removed from the text; no blank is introduced in place of the comment. This feature is often used to catenate two pieces of text; such usage is not portable.

Thus, in "*foo/* comment */bar*", the output will be "foobar" unless either "foo" or "bar" is a macro name. Expansion of "foo" and "bar" will take place if they are macro names. No macro expansion of "foobar" will be performed when it does not occur in a macro definition.

OTHER CONSIDERATIONS

The following features are not portable and are supported only with the *-pcc* flag.

Formal macro parameters are recognized in *#define* token strings, even inside character constants and quoted strings. The output from:

```
#define foo(a) '\a'
foo(bar)
```

is the six characters `'\bar'`.

Macro names are not recognized inside character constants or quoted strings during the regular scan. Thus:

```
#define foo bar
printf("foo");
```

does not expand "foo" in the second line, because it is inside a quoted string that is not part of a *#define* or *#undef*.

A mismatch between the number of formals and actuals in a macro call produces only a warning, not an error. Excess actuals are ignored; missing actuals are turned into null strings.

SEE ALSO

cc(1)

CONVEX C Guide

American National Standard for Information Systems — Programming Language C Document Number: X3J11/90-013

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*

B. W. Kernighan and D. M. Ritchie, *The C Programming Language, Second Edition*

NAME

lint – a C program verifier

SYNOPSIS

lint [*option* ...] *files* ...

DESCRIPTION

lint attempts to detect features of the C program *files* that are likely to be bugs, or non-portable, or wasteful. It also checks the type usage of the program more strictly than the compilers. Among the things that are currently found are unreachable statements, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions which return values in some places and not in others, functions called with varying numbers of arguments, and functions whose values are not used.

By default, it is assumed that all the *files* are to be loaded together; they are checked for mutual compatibility. Function definitions for certain libraries are available to *lint*; these libraries are referred to by a conventional name, such as '–lc', in the style of *ld*(1). Arguments ending in *.ln* are also treated as library files.

COMPATIBILITY MODES

CONVEX *lint* operates in one of four compatibility modes.

-ext This mode provides an extended implementation of ANSI C and an ANSI C and POSIX compatible library system. This is the default.

-std This option causes *lint* to enforce syntax in accordance with the ANSI C language definition. Certain extensions, notably the *long long* type, are not available in this mode. External functions are checked against ANSI C and POSIX P1003.1 conforming libraries. Many extensions provided by the default library system are not available.

-str This option causes *lint* to enforce syntax in accordance with the ANSI C language definition and to attempt to detect language features that cause it not to conform to ANSI C. Only the library functions defined by ANSI C are available.

-pcc Forces language and library interpretation based on the the original Kernighan and Ritchie definition, the Common C Compiler, and traditional Unix systems. This mode is compatible with previous versions of lint shipped by CONVEX.

PREPROCESSOR CONTROL OPTIONS

lint predefines the symbol *__lint*. In the backward-compatible mode the symbol *lint* is also predefined.

-D

-I

-U These options of *cc*(1) are recognized with the same meanings they have for the C compiler.

DIAGNOSTIC CONTROL

–d *name*

–d *name=e*

–d *name=w*

The *-d* option provides low level control of diagnostic output. The form *-d name* suppresses the named message. The form *-d name=w* causes the named message to be reported as a warning. The form *-d name=e* causes the named message to be reported as an error.

The following *names* can be used to control specific messages. The CONVEX C documentation on C compiler error messages provides examples for each of these diagnostic options.

arg_ptr_qual — Detects actual function parameters that do not have the same type qualifiers as those declared in the function prototype.

arg_ptr_ref — Detects actual function parameters that are not the same pointer type as those declared in the function prototype.

assign_in_condition — Detects assignment expressions in locations where conditional expressions are expected, such as *for* statements and *if* statements.

char_cvt_truncates — Indicates that a *char* data type is the target of a cast of an integer type larger than a *char* data type.

class_ignored — Indicates that an explicit storage class is used to declare a *struct* tag, *union* tag, *enum* tag, or *enum* member.

const_condition — Detects constant expressions in locations where conditional expressions are expected such as *if* and *switch* expressions.

const_not_init — Detects uninitialized constant variables.

cvt_changes_sign	Detects when an unsigned integer is assigned to a signed integer of the same size.
cvt_to_unsigned	Detects when an unsigned integer is assigned to a signed integer of the same size.
division_by_zero	Detects compile time division by zero.
dollar_names	Detects identifiers embedded with the $ character.
escape_range_sequence	Detects when an escape sequence in an integer constant is greater than 0xff.
eval_order	Detects when an expression has an undefined evaluation order.
float_suffix	Detects when the floating-point suffixes, f, F, l, or L, are used. This option has no effect in the strict and conforming compatibility modes.
function_parameter	Reports when a function is used as a function parameter; only function pointers can be used as a function parameter.
function_pointer_cast	Looks for non-function pointers that are assigned to function pointers.
hidden_arg	Indicates that the parameter of a function is hidden by an identifier with the same name declared in the outermost block of that function. This diagnostic affects only the backward-compatible mode.
hidden_extern	Indicates that a declaration using *extern* inside a function definition causes an identifier not in the block to be obscured. This diagnostic affects only the backward-compatible mode.
hides_outer	Indicates that an identifier prevents access to an identifier of the same name in an enclosing block.
implicit_decl	Detects when an implicit declaration is used because a function has not been declared previously.
incomplete_record	Indicates that a *union* or a *struct* was declared without any members.

int_cvt_truncates	Indicates that a *long long* variable is converted to a variable of type *int*.
integer_overflow	Detects when an integer constant larger than 64 bits is used.
long_long_suffix	Detects when the integer literal suffixes LL or `ll` are used. These suffixes are permitted only in the extended and backward-compatible modes because the *long long* data type is not available in the strict and conforming compatibility modes.
misplaced_lint_directive	Checks for a *lint* directive used in the wrong context.
neg_shift	Checks for right operands of shift operators that are negative constants.
negative_to_uns	Indicates when a negative constant is assigned to an unsigned type.
no_arg_type	Checks for function arguments declared without a data type.
no_external_declaration	Detects when no declarations are accessible to other compilation units.
non_int_bit_field	Check for bit fields that have type other than *int* or *unsigned int*.
nothing_declared	Detects empty declarations such as *int;*.
pointer_alignment_efficiency	Checks for operations that can result in inefficient memory alignment.
pp_argcount	Indicates that the number of arguments in the actual argument list of a macro does not match the number of arguments in the formal argument list.
pp_argsended	Indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis.
pp_badstr	Indicates incorrect use of #, the stringizing operator that is used in macro definitions.
pp_badtp	Indicates that the result of combining the left and right operands of the ## macro definition operator is illegal.

pp_extra	Indicates that a preprocessor directive has extra arguments. One common example is when the *# endif* directive is followed by text.
pp_idexpected	Indicates that a *#ifdef* or *#ifndef* does not have a legal identifier as an argument, or a formal argument of a function-like macro is not a legal identifier.
pp_line_range	Indicates the argument for the *#line* preprocessor directive does not have a value between 1 and 32767, inclusive.
pp_macro_arg	Indicates that two or more formal macro arguments have the same name.
pp_macro_redefinition	Indicates that the replacement list for a function-like macro is not the same as its original definition. The preprocessor uses the most recent replacement list.
pp_macro_redefinition_cmdl	Indicates that the replacement list for a function-like macro is not the same as its original definition on the command line. The preprocessor uses the most recent replacement list.
pp_malformed_directive	Indicates that the proper syntax for a preprocessor directive was not used.
pp_old_dir	Indicates that the comment style syntax for a compiler directive (pragma) was used.
pp_parse	Indicates that a `#if` directive has an invalid integer expression.
pp_undef	Indicates that the argument of the `#undef` directive is not permitted.
pp_undef_cmdl	Indicates that you attempted to undefined a macro that cannot be undefined on the command line.
pp_unrecognized_directive	Detects a preprocessor directive that the preprocessor does not recognize.
pp_unrecognized_pragma	Detects a pragma that the compiler does not recognize.
set_but_not_used	Detects variables that are assigned a value but are not used.

shift_too_large	Checks for constant right operands of shift operands that are too large.
short_cvt_truncates	Indicates that an integral type larger than a `short` integer is being assigned to a `short` integer.
skip_to_char	This diagnostic indicates that the compiler is skipping to a character to recover from an error.
skip_to_eof	This diagnostic indicates that the compiler must skip to the end of a file to recover from an error.
strict_syntax	Detects the lack of a semicolon after the last member in a `struct` or `union` declaration or the presence of a comma after the last enumeration constant in an `enum` declaration list.
uns_compare_neg	Indicates a comparison between a `short unsigned` integer and a negative constant.
uns_compare_zero	Indicates a comparison between an unsigned integer and zero.
unsigned_suffix	Detects use of the integer literal suffixes *U* or *u*. This option is not effective with the =*e* setting.
varargs_on_proto	Checks for usage of *lint* directive *VARARGS* on a function that has a function prototype. The ellipsis operator should be used instead.
varargs_too_large	Reports an error condition when the *lint VARARGS* argument is greater than the number of formal arguments in the function it precedes.
void_pointer_cast	Indicates that a pointer to `void` is involved in a cast operation.

CONTROL OPTIONS

The following options are used to control the types of checks performed by *lint*.

-a Report assignments of *long* values to *int* variables.

-b Report *break* statements that cannot be reached. (This is not the default because, unfortunately, most *lex* and many *yacc* outputs produce dozens of such comments.)

-c Complain about casts which have questionable portability.

-h Apply a number of heuristic tests to attempt to detect bugs, improve style, and reduce waste.

-n Do not check compatibility against the standard library.

-u Do not complain about functions and variables used and not defined, or defined and not used (this is suitable for running *lint* on a subset of files out of a larger program).

-v Suppress complaints about unused arguments in functions.

-x Report variables referred to by *extern* declarations, but never used.

-z Do not complain about structures that are never defined (e.g., using a structure pointer without knowing its contents.).

MISCELLANEOUS OPTIONS

–B*dir* Finds substitute lint (*cpp, lint1, lint2,* and *errmsgc*) in the directory *dir.* If *dir* is not specified, the standard backup version in the directory */usr/lib/oldcc* is used instead. For example, the command `lint -B/usr/new` would invoke */usr/new/lint1* instead of the default */usr/lib/cc/lint1.*

-C Used to create lint libraries. If *files* are the C sources of a library *congress,* the command

lint –C*congress files* . . .

would create a lint library *llib-lcongress.ln* which would be suitable for "linting" programs using the library *congress.*

–fi Translate floating-point constants to IEEE format and perform floating-point operations in IEEE mode. This option requires that the machine be equipped with IEEE floating-point hardware. If no floating-point format is specified, the site default is used. Arithmetic performed under this option does not conform to the IEEE standard.

–float sp_ops

–float dp_ops

–float sp_const

–float dp_const

These options control the floating-point system used by *lint* and are provided for compatibility with *cc. sp_ops* and *dp_ops* control the precision of the operations performed on float operands. When *sp_ops* is specified, 32 bit operations are performed. When *dp_ops* is specified, the float operands are converted to double and 64 bit operations are performed.

sp_const and *dp_const* control the representation of unsuffixed floating-point constants (which are normally represented in double precision). *sp_const* causes these constants to be represented in single precision. Some loss of accuracy may result. *dp_const* explicitly invokes the default.

sp_ops is the default in all but the backward-compatible mode.

dp_const is the default in all modes.

–fn Translate floating-point constants to native CONVEX format and perform floating-point operations in native mode. If no floating-point format is specified, the site default is used.

–sso Treat the result of the sizeof operator as a signed value. The default is unsigned int. This is only available in the backward-compatible mode.

–tl *time*
Sets the maximum CPU time limit on *lint* to *time* minutes. If the CPU time exceeds the allotted time, *lint* is aborted.

–vn Identifies *lint* version. Outputs the version numbers of *lint, cpp, lint1, lint2* and *errmsgc* to *stderr.*

SOURCE LEVEL CONTROL

Certain conventional comments in the C source will change the behavior of *lint*:

/*NOTREACHED*/
At appropriate points, stops comments about unreachable code.

/*VARARGS*n**/
Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0. Should only be used with non-prototyped function definitions.

/*ARGSUSED*/
Turns on the **–v** option for the next function.

/*LINTLIBRARY*/
At the beginning of a file, shuts off complaints about unused functions in this file.

ENVIRONMENT VARIABLES

CONVEX *lint* prepends the value of the environment variable **LINTOPTIONS** (if it is set) to each command line so that options need not be specified every time *lint* is invoked. For example,

```
setenv LINTOPTIONS '-d pointer_alignment_efficiency'
```

causes *lint* to not report pointer casts which could result in inefficiently aligned pointers and

```
setenv LINTOPTIONS '-pcc'
```

causes *lint* operate in the backward-compatible mode.

The preprocessor *cpp* has a similar variable **CPPOPTIONS** which can be used to affect its behavior when it is invoked directly. However, **CPPOPTIONS** does not have any effect when *lint* invokes *cpp* (lint removes the variable before invoking *cpp*).

FILES

/usr/bin/lint	lint driver
/usr/lib/cc/errmsgc	error message file
/usr/lib/cc/lint[1|2]	programs
/usr/lib/lint/llib-lc.ln	declarations for standard functions
/usr/lib/lint/llib-lc	human readable version of above
llib-l*.ln	library created with –**C**

SEE ALSO

cc(1)
"Lint, a C Program Checker" in the *ConvexOS Tutorial Papers*

BUGS

There are some things you just **can't** get *lint* to shut up about.

exit(2) and other functions which do not return are not understood; this causes various lies.

The library definition for -std and -str modes is the same as that of the -ext mode (see COMPATIBILITY MODES section).

NAME

intro – introduction to Cray compatible bit manipulation library functions

DESCRIPTION

These functions constitute the C bit manipulation library, *libbint*. The link editor searches this library under the "–lbint" option. Declarations for these functions may be obtained from the include file *<bint.h>*.

LIST OF FUNCTIONS

Name	*Appears on Page*	*Description*
_count	bitcount(3BIT)	count the number of "1" bits
_dshiftl	bitshift(3BIT)	rotate bits left or shift left-most bits of *source* into right-most bits of *target*
_dshiftr	bitshift(3BIT)	rotate bits right or shift right-most bits of *source* into left-most bits of *target*
_gbit	bitchange(3BIT)	get a single bit
_gbits	bitchange(3BIT)	get a consecutive sequence of bits
IBCLR	bitmil(3BIT)	clear a bit
IBITS	bitmil(3BIT)	get a consecutive sequence of bits
IBSET	bitmil(3BIT)	set a bit
_ldzero	bitcount(3BIT)	count the number of leading zeros
_leadz	bitcount(3BIT)	count the number of leading zeros
_mask	bitmask(3BIT)	create a mask of right or left justified "1" bits
_maskl	bitmask(3BIT)	create a mask of left justified bits
_maskr	bitmask(3BIT)	create a mask of right justified bits
MVBITS	bitmil(3BIT)	move a consecutive sequence of bits
_parity	bitcount(3BIT)	determine the parity
_pbit	bitchange(3BIT)	replace a single bit
_pbits	bitchange(3BIT)	replace a consecutive sequence of bits
_popcnt	bitcount(3BIT)	count the number of "1" bits
_poppar	bitcount(3BIT)	determine the parity

DIAGNOSTICS

Some of the functions can be accessed using faster function-like macros. These macros are called by default in the extended and backward-compatible modes. These macros offer increased speed because they call intrinsic functions.

To prevent the intrinsic functions from being linked into your program, compile with the
-D__NO_INLINE_BINT command line option. You could also compile your program with the
-D__NO_INLINE command line option which disables all function-like macros.

When an error occurs, the *errno* variable is not set.

FILES

/usr/lib/libbint.a
/usr/include/bint.h

SEE ALSO

bitchange(3BIT), bitmil(3BIT), bitcount(3BIT), bitmask(3BIT), bitshift(3BIT), bint.h(3BIT)

NAME

bint.h, __NO_INLINE_BINT, __NO_INLINE, bint_t – header file contains declarations and function prototypes for Cray-compatible C bit manipulation functions.

SYNOPSIS

#include <bint.h>

typedef long long bint_t;

DESCRIPTION

bint.h contains declarations for C bit manipulation functions. Most of the functions can be accessed using faster function-like macros. If you want to call the function instead of the function-like macro, compile with the *-D__NO_INLINE_BINT* or the *-D__NO_INLINE* command-line options. The latter eliminates use of all function-like macros when a choice between a function or a function-like macro exists. Compile your program with the *-lbint* option at the end of the *cc* command-line; this option tells the linker that the libbint.a library contains the executable form of the bit manipulation functions.

bint_t is the data type returned by all of the bit manipulation functions.

COMPATIBILITY

Because the functions use an argument of type *long long*, only the extended and backward-compatible compatibility modes can be used with the bit manipulation functions. The extended compatibility mode is the default mode of the compiler.

The bit manipulation functions implemented by CONVEX are depend on type coercion to match the actual function parameters with the formal function parameters. If you include the < *bint.h* > header file in your program, the arguments will be coerced to the type expected by the library routines.

This coercion may cause a problem with the *leadz* and *ldzero* functions which count the number of leading zeros in their argument. If the argument is a *char*, there will be 56 more zeros than expected because the *char* data type is coerced from an 8-bit value to a 64-bit value.

FILES

/usr/lib/libbint.a
/usr/include/bint.h

SEE ALSO

The following man pages describe the Cray compatible bit manipulation functions declared in the *bint.h* header file:

bitchange(3BIT),
bitcount(3BIT),
bitmask(3BIT),
bitmil(3BIT),
bitshift(3BIT).

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

NAME

_gbit, _gbits, _pbit, _pbits – Cray-compatible bit manipulation functions that get or set the value of a particular group of bits.

SYNOPSIS

```
#include <bint.h>

bint_t _gbit(bint_t target, int i);

bint_t _gbits(bint_t target, int length,
              int i);

bint_t _pbit(bint_t target, int i,
             bint_t source);

bint_t _pbits(bint_t target, int length,
              int i, bint_t source);
```

DESCRIPTION

_gbit returns the value in *target* of the bit located in the *i*th bit position from the right.

_gbits returns the value in *target* consisting of *length* bits beginning with the *i*th bit position from the right. The bits are right justified.

_pbit returns the value of *target* in which the bit located in the *i*th bit position from the right is replaced by the least significant bit of *source*.

_pbits returns the value of *target* in which *length* bits beginning with the *i*th bit position from the right are replaced by the value of the *length* least significant bits of *source*.

FILES

/usr/include/bint.h
/usr/lib/libbint.a

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

NAME

_count, _ldzero, _leadz, _parity, _popcnt – Cray-compatible bit manipulation functions that perform population count, leading zero count, and parity count.

SYNOPSIS

#include <bint.h>

bint_t _count(bint_t target);

bint_t _ldzero(bint_t target);

bint_t _leadz(bint_t target);

bint_t _parity(bint_t target);

bint_t _popcnt(bint_t target);

bint_t _poppar(bint_t target);

DESCRIPTION

_count and *_popcnt* return the number of "1" bits contained in *target*.

_ldzero and *_leadz* return the number of leading zeros contained in *target*. Remember that this argument is a *long long* data type, or is coerced to the *long long* data type.

_parity and *_poppar* return the parity of *target*: zero if it has an even number of "1" bits or one if it has an odd number of "1" bits.

FILES

/usr/include/bint.h
/usr/lib/libbint.a

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

EXAMPLE

The following example computes the population count, the leading zero count, and the parity count of a *char* data type:

```
#include <stdio.h>
#include <bint.h>

main()
{
  unsigned char arg = 0x30;
  bint_t retval;

  retval = _count( arg );
  (void) printf("population count = %lld\n", retval );
  retval = _leadz( arg );
  (void) printf("leading zeros = %lld\n", retval-56 );
  retval = _parity( arg );
  (void) printf("parity = %lld\n", retval );
}
```

Note that the return value of the *_leadz* function is subtracted by 56. This is to account for the extra zeros introduced by the coercion to the *long long* date type.

NAME

_mask, _maskl, _maskr – Cray-compatible bit manipulation functions that create a mask of contiguous "1" bits.

SYNOPSIS

#include <bint.h>

bint_t _mask(bint_t length);

bint_t _maskl(bint_t length);

bint_t _maskr(bint_t length);

DESCRIPTION

_mask returns a mask of *length* left-justified "1" bits if 0 ≤ *length* ≤ 63. Otherwise, it returns a mask of 128-*length* right-justified "1" bits.

_maskl returns a mask of *length* left-justified "1" bits.

_maskr returns a mask of *length* right-justified "1" bits.

FILES

/usr/include/bint.h
/usr/lib/libbint.a

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

BUGS

The *_mask* function does not verify that *length* is a constant between 0 and 128.

The *_maskl* and *_maskr* functions do not verify that *length* is a constant between 0 and 64.

NAME

IBITS, IBSET, IBCLR, MVBITS – MIL standard bit manipulation function-like macros.

SYNOPSIS

#include <bint.h>

bint_t IBITS(bint_t target,int length,int i);

bint_t IBSET(bint_t target, int i);

bint_t IBCLR(bint_t target, int i);

bint_t MVBITS(bint_t source, int source_i, int length, bint_t target, int target_i);

DESCRIPTION

The MIL standard bit manipulation function-like macros, *IBITS*, *IBSET*, *IBCLR*, and *MVBITS* are implemented using the C bit manipulation functions, *_gbits*, *_pbit*, and *_pbits*. There are no corresponding functions with these names; they can only be used by inclusion of the <bint.h> include file. These function-like macros are only available in the extended and backward-compatible compatibility modes of the compiler. The extended compatibility mode is the default mode of the compiler.

The code for these function-like macros can be linked into your program when you include the *-lbint* option on the *cc* command line.

RETURN VALUES

IBITS returns the value in *target* consisting of *length* bits beginning with the *i*th bit position from the right. The bits are right-justified.

IBSET returns the value of *target* in which the bit located in the *i*th bit position from the right is set to a "1" bit.

IBCLR returns the value of *target* in which the bit located in the *i*th bit position from the right is cleared with a "0" bit.

MVBITS moves *length* bits starting at the *source_i*th bit position (from the right) in *source* into the bits starting at the *target_i*th bit position (from the right) in *target*.

FILES

/usr/include/bint.h
/usr/lib/libbint.a

SEE ALSO

bitchange(3BIT)

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

NAME

_dshiftl, _dshiftr – Cray-compatible bit manipulation functions that shift bits between two arguments, perform a rotate-right shift, or perform a rotate-left shift.

SYNOPSIS

```
#include <bint.h>

bint_t _dshiftl(bint_t target,          bint_t source, int length);

bint_t _dshiftr(bint_t source,          bint_t target, int length);
```

DESCRIPTION

_dshiftl returns the value of target in which its *length* rightmost bits are replaced by the *length* leftmost bits of *source*. If *source* and *target* have the same value, a rotate-left occurs.

_dshiftr returns the value of target in which its *length* leftmost bits are replaced by the *length* rightmost bits of *source*. If *source* and *target* have the same value, a rotate-right occurs.

FILES

/usr/include/bint.h
/usr/lib/libbint.a

DIAGNOSTICS

No domain or range checking is performed on these functions. *errno* is not set when an error occurs.

APPENDIX G
Error Messages

This appendix describes error messages encountered when the `cc` or the `lint` programs process C source files. The first part of this appendix discusses the control of some error messages; the second part details the error messages and, in some cases, includes a short example that shows the cause of the error message.

Error Message Control

Compatibility Modes

The compiler generates error messages specific to each compatibility mode. For example, the backward-compatible mode does not allow use of prototypes. Choose the compatibility mode required for your application. Refer to Chapter 3 for more information on compatibility modes.

Compiler Diagnostic Options

Many compiler options control the diagnostic output of the compiler. For example, `-w` suppresses all warning messages.

The `-d` option can convert a warning message to an error message. This conversion is available for only a certain number of diagnostic messages. The syntax for this option is:

-d *name*[={w|e}]

where *name* is the name of the diagnostic message, `=w` converts *name* to a warning message, `=e` converts *name* to an error message, and no suffix after *name* suppresses the *name* diagnostic message.

Error messages prevent the creation of an executable program; warning messages do not. For example, to permit extra characters after the `#endif` preprocessor directive, compile with:

```
-d pp_extra
```

or to generate a error message, use:

```
-d pp_extra=e
```

The default for this error is a warning message.

Note	Converting an error message to a warning message or removing it entirely can cause undefined behavior in the compiler. Care should be used when overriding error messages.

The diagnostic messages you can override are listed below. They are accompanied by a description of the condition that they control. If the condition exists, a message is generated only if the message is converted to a warning or error message. Either of these cases might be the default. *Removing a diagnostic message does not affect the behavior of the compiler; it merely prevents the message from being displayed.* Unless otherwise noted, all diagnostic options are available with both `cc` and `lint`. None of these diagnostic options is available with `/lib/cpp`.

`arg_ptr_qual`
: Detects actual function parameters that do not have the same type qualifiers as those declared in the function prototype.

`arg_ptr_ref`
: Detects actual function parameters that are not the same pointer type as those declared in the function prototype.

`assign_in_condition`
: Detects assignment expressions in locations where conditional expressions are expected.

`char_cvt_truncates`
: Indicates that a `char` data type is the target of a cast of an integer type larger than a `char` data type. This is available only in `lint`.

`class_ignored`
: Indicates that an explicit storage class is used to declare a `struct` tag, `union` tag, `enum` tag, or `enum` member.

`const_condition`
: Detects constant expressions in locations where conditional expressions are expected such as `if` and `switch` expressions. This is available only in `lint`.

`const_not_init`
: Detects uninitialized constant variables.

`cvt_changes_sign`
: Detects when an unsigned integer is assigned to a signed integer of the same size. This is available only in `lint`.

`cvt_to_unsigned`
: Detects when a signed integral type is assigned to an unsigned integral type. The unsigned integral type can be the same size or larger than the signed integral type. This is available only in `lint`.

`division_by_zero`
: Detects compile time division by zero.

`dollar_names`
: Detects identifiers embedded with the $ character.

`escape_range_sequence`
: Detects when an escape sequence in an integer constant is greater than 0xff.

`eval_order`	Detects when an expression has an undefined evaluation order. This is available only in `lint`.
`float_suffix`	Detects when the floating-point suffixes, f, F, l, or L, are used. This option has no effect in the strict and conforming compatibility modes.
`function_parameter`	Reports when a function is used as a function parameter; only function pointers can be used as a function parameter. This is on by default in `lint`.
`function_pointer_cast`	Looks for non-function pointers that are assigned to function pointers. This is available only in `lint`.
`hidden_arg`	Indicates that the parameter of a function is hidden by an identifier with the same name declared in the outermost block of that function. This diagnostic affects only the backward-compatible mode of the compiler.
`hidden_extern`	Indicates that a declaration using `extern` inside a function definition causes an identifier not in the block to be obscured. This diagnostic affects only the backward-compatible mode of the compiler.
`hides_outer`	Indicates that an identifier prevents access to an identifier of the same name in an enclosing block.
`implicit_decl`	Detects when an implicit declaration is used because a function has not been declared previously.
`incomplete_record`	Indicates that a `union` or a `struct` was declared without any members. This is available only in `lint`.
`int_cvt_truncates`	Indicates that a `long long` variable is converted to a variable of type `int`. This is available only in `lint`.
`integer_overflow`	Detects when an integer constant larger than 64 bits is used.
`long_long_suffix`	Detects when the integer literal suffixes LL or `ll` are used.
`misplaced_lint_directive`	Checks for a `lint` directive used in the wrong context. This is available only in `lint`.
`neg_shift`	Checks for right operands of shift operators that are negative constants. This is available only in `lint`.
`negative_to_uns`	Indicates when a negative constant is assigned to an unsigned type.
`no_arg_type`	Checks for function arguments declared without a data type.
`no_external_declaration`	Detects when no declarations are accessible to other compilation units.
`non_int_bit_field`	Checks for types of bit fields other than `int` or `unsigned int`.

`nothing_declared` — Detects empty declarations such as `int;`.

`null_effect_expression` — Detects intrinsic math function calls that have no effect. This option is available only with `cc`.

`pointer_alignment_efficiency` — Checks for operations that can result in inefficient memory alignment. This is available only in `lint`.

`pp_argcount` — Indicates that the number of arguments in the actual argument list of a macro does not match the number of arguments in the formal argument list.

`pp_argsended` — Indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis.

`pp_badstr` — Indicates incorrect use of #, the stringizing operator that is used in macro definitions.

`pp_badtp` — Indicates that the result of combining the left and right operands of the ## macro definition operator is illegal.

`pp_extra` — Indicates that a preprocessor directive has extra arguments.

`pp_idexpected` — Indicates that a `#ifdef` or `#ifndef` does not have a legal identifier as an argument, or a formal argument of a function-like macro is not a legal identifier.

`pp_line_range` — Indicates the argument for the `#line` preprocessor directive does not have a value between 1 and 32,767, inclusive.

`pp_macro_arg` — Indicates that two or more formal macro arguments have the same name.

`pp_macro_redefinition` — Indicates that the replacement list for a function-like macro is not the same as its original definition. The preprocessor uses the most recent replacement list.

`pp_macro_redefinition_cmdl` — This message indicates that a macro was redefined with a new definition on the command line. This is available only in `lint`.

`pp_malformed_directive` — Indicates that the proper syntax for a preprocessor directive was not used.

`pp_old_dir` — Indicates that the comment style syntax for a compiler directive (pragma) was used.

`pp_parse` — Indicates that a `#if` directive has an invalid integer expression.

`pp_undef` — Indicates that the argument of the `#undef` directive is not permitted.

`pp_undef_cmdl` — Indicates that you attempted to undefine a macro that cannot be undefined on the command line. This is available only in `lint`.

pp_unrecognized_directive	Detects a preprocessor directive that the preprocessor does not recognize.
pp_unrecognized_pragma	Detects a pragma that the compiler does not recognize.
set_but_not_used	Detects variables that are assigned a value but are not used. This is available only in lint.
shift_too_large	Checks for right operands of shift operands that are too large. This option only examines operands that are constants. This is available only in lint.
short_cvt_truncates	Indicates that an integral type larger than a short integer is being assigned to a short integer. This is available only in lint.
skip_to_char	This diagnostic indicates that the compiler is skipping to a character to recover from an error.
skip_to_eof	This diagnostic indicates that the compiler must skip to the end of a file to recover from an error.
strict_syntax	Detects the lack of a semicolon after the last member in a struct or union declaration or the presence of a comma after the last enumeration constant in an enum declaration list.
uns_compare_neg	Indicates a comparison between a short unsigned integer and a negative constant. This is available only in lint.
uns_compare_zero	Indicates a comparison between an unsigned integer and zero. This is available only in lint.
unsigned_suffix	Detects use of the integer literal suffixes U or u. This option is not effective with the =e setting.
varargs_on_proto	Checks for use of lint directive VARARGS on a function that has a function prototype. The ellipsis operator should be used instead. This is available only in lint.
varargs_too_large	Reports an error condition when the lint VARARGS argument is greater than the number of formal arguments in the function it precedes. This is available only in lint.
void_pointer_cast	Indicates that a pointer to void is involved in a cast operation. This is available only in lint.

Default settings for these diagnostic conditions in the four compatibility modes of CONVEX C are listed in Table G-1.

The "-" character indicates that the default for the diagnostic condition is to generate no message. The "x" character indicates that the diagnostic has no affect in the respective mode.

Table G-1: Diagnostic Condition Default Settings

Name	Compatibility Mode			
	-ext	-std	-str	-pcc
arg_ptr_qual	warn	warn	warn	warn
arg_ptr_ref	warn	warn	warn	warn
assign_in_condition	-	-	-	-
char_cvt_truncates	-	-	-	-
class_ignored	warn	warn	warn	-
const_condition	-	-	-	-
const_not_init	warn	warn	warn	warn
cvt_changes_sign	-	-	-	-
cvt_to_unsigned	-	-	-	-
division_by_zero	warn	warn	warn	warn
dollar_names	-	warn	warn	-
escape_range_sequence	-	warn	warn	-
eval_order	warn	warn	warn	warn
float_suffix	-	x	x	-
function_parameter	-	-	-	-
function_pointer_cast	warn	warn	warn	warn
hidden_arg	x	x	x	warn
hidden_extern	x	x	x	warn
hides_outer	-	-	-	-
implicit_decl	-	-	-	-
incomplete_record	-	-	-	-
int_cvt_truncates	-	-	-	-
integer_overflow	-	warn	warn	-
long_long_suffix	-	warn	warn	-
misplaced_lint_directive	warn	warn	warn	warn
neg_shift	warn	warn	warn	warn
negative_to_uns	-	-	-	-
no_arg_type	-	-	-	-
no_external_declaration	-	warn	warn	-
non_int_bit_field	warn	warn	warn	warn
nothing_declared	warn	warn	warn	-
null_effect_expression	warn	warn	warn	warn
pointer_alignment_efficiency	-	-	-	-
pp_argcount	warn	warn	warn	warn
pp_argsended	error	error	error	error
pp_badstr	warn	warn	warn	warn
pp_badtp	warn	warn	warn	warn
pp_extra	warn	warn	warn	warn
pp_idexpected	error	error	error	error
pp_line_range	error	error	error	error
pp_macro_arg	warn	warn	warn	warn

Table G-2: Diagnostic Condition Default Settings (cont)

Name	Compatibility Mode			
	-ext	-std	-str	-pcc
pp_macro_redefinition	warn	warn	warn	warn
pp_macro_redefinition_cmdl	warn	warn	warn	warn
pp_malformed_directive	error	error	error	error
pp_old_dir	-	-	-	-
pp_parse	error	error	error	error
pp_undef	error	error	error	error
pp_undef_cmdl	warn	warn	warn	warn
pp_unrecognized_directive	error	error	error	error
pp_unrecognized_pragma	warn	warn	warn	warn
set_but_not_used	warn	warn	warn	warn
shift_too_large	warn	warn	warn	warn
short_cvt_truncates	-	-	-	-
skip_to_char	warn	warn	warn	warn
skip_to_eof	warn	warn	warn	warn
strict_syntax	warn	warn	warn	-
uns_compare_neg	-	-	-	-
uns_compare_zero	-	-	-	-
unsigned_suffix	-	-	-	error
varargs_on_proto	warn	warn	warn	warn
varargs_too_large	warn	warn	warn	warn
void_pointer_cast	-	-	-	-

Error Message Catalog

The error messages are listed in the following pages in alphabetical order. In determining the keyword used to sort the messages, the following words were ignored:

- a
- an
- cc:
- lint:
- the
- this
- Words in *italics*
- Words in quotation marks

Each message description includes:

- The message generated by the compiler
- The message name if it can be used with the -d compiler option
- A short explanation of the error.

Error Messages

`cc: actual and formal point to different types`

Message Name: `arg_ptr_ref`

Actual function parameters are parameters used in a function call, while formal function parameters are parameters declared in a function prototype or function definition. This error message occurs when the pointer declared in a function prototype is not compatible with the pointer passed to the function.

Example:

```
/* compile with cc -d arg_ptr_ref=e file.c */
extern int func1(int *c);

main()
{
     int a;
     float  *b;

     a = func1( b );
}
```

In this example, the actual parameter is a pointer to `float` and the formal parameter is a pointer to `int`.

If you must pass different pointer types to a function, declare the formal parameter type as a `void` pointer type as in the following example.

Example:

```
/* compile with cc -d arg_ptr_ref=e file.c */
extern int func1(void *c);

main()
{
     int a;
     int *b;

     a = func1( b );
}
```

A `void` pointer is a generic pointer compatible with any pointer type.

`cc: ambiguous old form assignment operator:` *operator* `interpreted as` *operator*

Some non-ANSI C compilers permit assignment operators of the form =*op*, where *op* could be one of `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, or `|`. Use of these operators can lead to ambiguous situations. For example,

```
x =- 7;
```

could be either "assign -7 to x," or "decrement x by 7."

These operators are obsolete in ANSI C. The message indicates what assumption the compiler made when such an operator was encountered.

`cc: argument` *name* `unused in function` *func*

This message indicates that function *func* declares *name* as a function parameter, but does not use it in its function definition.

Example:

```
int func( int arg1, int arg2 )
{
    return (arg1);
}
```

Correct this error by removing the unused function parameter from the function definition. Be sure that all function prototypes for this function remain consistent with the function definition.

`cc: argument` *'arg'* `is hidden by declaration in outermost block at line` *line* `of` *file*

Message Name: `hidden_arg`

This message is only generated with source files compiled in the backward-compatible mode. It indicates that the parameter *arg* is hidden by an identifier with the same name declared in the outermost block of that function.

Example:

```
/* compile with cc -pcc -d hidden_arg=w file.c */
f( a )
int a;
{
    float a; /* Hides parameter a */
}
```

Correct this error by renaming one of the identifiers and all references to it.

`cc: array declared with zero or negative number of elements.`

When an array is declared, the expression delimited by [and] must evaluate to a positive integral constant. If not, the compiler cannot allocate any storage for the array.

`cc: array element type has unknown size`

This message occurs when an array is declared with an incomplete type. Incomplete types do not provide enough information to enable a compiler to determine their size.

Example:

```
struct some_s array[10];
```

This code generates this message if no `struct` has been defined with the tag `some_s`. This condition is not reported in the backward-compatible mode of the compiler.

`cc: array has too many dimensions; limit is` *limit*.

An array was declared with too many dimensions. The maximum number of dimensions is *limit*.

`cc: array is too big`

The number of elements in the array exceeds the maximum size of an array. Replace the array with smaller arrays to correct this error.

`cc: array of functions illegal.`

An array of functions is illegal, but an array of pointers to functions is not. The syntax for an array of pointers to functions is

type `(*`*array-name*`[`*num*`])();`

where

type	is the data type returned by the functions.
array-name	is the array name.
num	is the number of elements in the array.

Example:

```
int *(*funcs[5])();
```

This example declares an array of five pointers to functions that each return a pointer to an integer.

```
cc: assignment operator occurs in conditional expression
```

Message Name: `assign_in_condition`

This message indicates that an assignment operation occurs in a context in which a conditional operation is expected. The following source code contains two examples that demonstrate this condition:

```
/* check with cc -d assign_in_condition=w file.c */
main()
{
    int a,i;

    for(i=1; i=0; i++) /* example 1 */
        /* null */;
    if(a = 3)          /* example 2 */
        return (a);
    return (1);
}
```

The first example detects a conditional operation in a `for` statement, and the second detects a conditional operation in an `if` statement. This may be intentional. If it is not, convert the assignment operator to the correct conditional operator.

```
cc: Attempt to modify an object with const-qualified type
```

Identifiers that are qualified by the `const` qualifier cannot be assigned a value, except when they are initialized.

Example:

```
const int x;
int y;

x = y;
```

This code generates this message. One way to initialize a variable with `const` qualified type is:

```
const int z = 5;
```

```
cc: automatic aggregates may not be initialized.
```

This message occurs only in the backward-compatible mode. An aggregate is a `struct` or `union` data type, and is automatic if its storage class is automatic. CONVEX C compilers prior to CONVEX C V4.0 did not permit automatic aggregates in block scope to be initialized. Consequently, they cannot be initialized in the backward-compatible mode.

`cc: bad argument 'argument' for directive directive`

argument is illegal when used with the pragma (directive) indicated in the error message. Refer to Appendix B, "Pragmas," for assistance with the pragma (directive).

`cc: begin_tasks directive with no end_tasks`

The optimization pragma **begin_tasks** tells the compiler to generate parallel code for the series of tasks that immediately follow. The **next_task** pragma marks the end of the preceding task and the start of another task. All such sequences of tasks must be terminated with the **end_tasks** pragma. These pragmas tell the compiler that certain nonloop sections of code can execute safely in parallel.

The following code fragment causes the compiler to generate this error message at optimization level -O3:

```
#pragma _CNX begin_tasks
task1();
#pragma _CNX next_task
task2();
```

Correct this error by appending the **end_tasks** pragma:

```
#pragma _CNX begin_tasks
task1();
#pragma _CNX next_task
task2();
#pragma _CNX end_tasks
;
```

`cc: a bit field must have type int or unsigned int`

Message Name: non_int_bit_field

A bit field that does not have type int or type unsigned int has been detected. The ANSI C standard permits only these two types in addition to signed int. The following code generates this error message:

```
/* compile with cc -d non_int_bit_field=e file.c */
struct s {
    signed char d : 5;
};
```

To permit other integral data types such as char or signed long, compile with cc -d non_int_bit_field file.c.

`cc: break statement not in do, for, switch, or while statement.`

`break` statements may occur only within a `do`, `for`, `switch`, or `while` statement. The following code

```
main()
{
    break;
    return(0);
}
```

is one example of this error. This error is often caused by misplaced { and } braces.

`cc: call error: formal and actual have different struct types.`

This message occurs when a `struct` declared in a function prototype is not the same as the `struct` passed in a function call.

Example:

```
struct tag1 { int z; };
struct tag2 { char x; };
extern void f( struct tag1 formal );

int g() {
    struct tag2 actual;
    f( actual );
}
```

In this example, the record structure that declares `actual` is not the same as the record structure that declares `formal`. The tags must be the same; if the contents of `tag2` were an `int` instead of a `char`, the message would still be generated. One way to correct this error is to declare the actual parameter with a `typedef` name instead of a `struct` declaration:

```
struct tag1 { int z; };
typedef struct tag1 tag1_t;
extern void f( struct tag1 formal );

int g() {
    tag1_t actual;
    f( actual );
}
```

`typedef` names are synonyms for the types that define them.

```
cc: call error: incompatible formal and actual types.
```

This message occurs when the data types of the actual parameters of a function do not agree with the data types of the formal definition or declaration of the same function.

Example:

```
void func(int a);

main()
{
    int *p;

    func(p);
}
```

In this example, the actual parameter is a pointer, while the formal parameter is an `int`. Similarly, if the actual parameter is a nonzero value and the formal parameter is a pointer, the message occurs. Correct the error by modifying the data type of the formal or actual function parameter.

`cc:` *'name'* `can not be redefined with -D, ignored`

This message indicates that you attempted to define a macro that cannot be redefined. The macros that cannot be redefined are `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__`, and `defined`.

`cc:` *macro* `can not be redefined, ignored`

This message indicates that a `#define` preprocessor directive has `__LINE__` or `__FILE__` as an argument. These macro constants cannot be redefined in any compatibility mode. Macro constants that cannot be redefined in the ANSI C compatibilities modes are `__DATE__`, `__TIME__`, and `__STDC__` (in the standard and strict modes).

`cc: can not initialize array` *'array'* `with elements of unknown size.`

This message occurs in the backward-compatible mode when an array, declared with an undefined `struct` or `union`, is initialized. For example:

```
struct unknown_s a[10] = { 3 };
```

Because the `unknown_s` has not been defined, the compiler cannot initialize the array. Correct this error by defining the `struct` or removing the initialization.

`cc: can not open file` *name*

This message is only generated when the C preprocessor is run by itself with the `cpp` command. It indicates that the preprocessor was not able to open the output file included on the command line. This message can occur if you invoke the preprocessor in a directory for which you do not have write permission or you do not have write permission on the indicated file.

`cc: can not use -> or . on type whose members are not defined.`

This message occurs when the left operand of the -> or . operator is declared with an incomplete `struct` or `union`. A structure is incomplete if its are not members declared; the compiler does not know the memory requirements of the structure. The following code generates this message:

```
void func()
{
    struct incomplete_s *p;
    p->unknown = 5;
}
```

Correct this error by defining the members of incomplete structures.

`Can't close file "`*source_file*`" - can't continue.`

`lint` creates an intermediate file when it processes your C source files. When `lint` is unable to close the intermediate file that it creates, it generates this message. The intermediate file may have been corrupted by another system process. Contact your system manager when this error occurs.

`Can't open file "`*file_name*`" - can't continue.`

The compiler is unable to open file *file_name*. One cause of this error is that the compiler is unable to find a source file.

`Can't recover from previous errors`

The compiler encountered too many errors to continue processing the source code. When the compiler reports an error, it makes an assumption about what the programmer intended before it continues compiling. If that assumption is incorrect, the compiler can report nonexistent errors. This *cascade effect* can prevent the compiler from checking the entire source file. Removing the initial error can eliminate subsequent errors.

```
cc: can't take the address of register variable 'var_name'.
```

This message occurs when the address-of operator (&) is applied to a variable that has been declared with the `register` storage class:

```
main()
{
    register int a;
    int *b;

    b = &a;
}
```

Correct this error by removing the `register` storage-class specifier from the variable declaration.

```
Can't Write to file "file_name" - can't continue.
```

The compiler is unable to open file *file_name* for writing. Check to make sure that the directory *file_name* file is in can be written to. Also check to make sure that you can write to the *file_name* file.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
cannot compile; machine serial number mismatch
```

The serial number of the CONVEX computer does not match the serial number embedded in the C compiler. Contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Refer to the contact(1) man page to learn how to contact the TAC.

```
cc: Cannot open CXdb data file 'name'.
```

The compiler stores data used by the CONVEX visual debugger in the .CXdb directory; this is a subdirectory of the current directory. When this message occurs, check to make sure that the compiler has write permission in the .CXdb directory.

```
cc: Cannot open CXdb data file directory.
```

This message indicates that the compiler cannot open the .CXdb directory in the current directory. Check to make sure that you have write permission in your current directory and that your umask value is appropriate.

```
cc: case label not constant or constant expression.
```

In the construct `case expression:`, `expression` must be a constant expression.

```
cc: case statement not in switch statement.
```

The `case` statement can only occur within the compound statement of C `switch` statements. This message often indicates misplaced braces.

cc: `character constant` *name* `is too long. Max length is` *length*.

The maximum length of a character constant is *length* characters. For example, if the maximum length of a character constant is 8 characters,

```
char ch = 'abcdefgh';
```

is a legal declaration of a character constant, but

```
char ch2 = '123456789';
```

has too many characters in its initialization. In the strict and conforming modes, the maximum length of a character constant is 4 characters, in the other compatibility modes the maximum length is 8 characters.

```
cc: Compile time division by zero detected
```

Message Name: `division_by_zero`

Division by zero is detected at compile time only when a constant with the value of zero appears in the denominator of a fraction.

Example:

```
/* compile with cc -d division_by_zero=e file.c */
int x = 3/0;
```

This code produces this message.

cc: Compile time integer overflow detected

Message Name: `integer_overflow`

An integer overflow is detected at compile time when an integral constant requires more than 64 bits of storage exclusive of the data type.

Example:

```
/* compile with cc -d integer_overflow=e file.c */
char x = 0xfffffffffffffff1;
```

This code produces this message.

>>>>> C O M P I L E R E R R O R <<<<<
>>>>> See your system manager for help <<<<<
cc: Compiler error on line *linenum* of *filename*.

An internal compiler error occurred. Contact your system manager or the CONVEX Technical Assistance Center (TAC). Refer to the contact(1) man page to learn how to contact the TAC.

filename is the name of a file used to create the C compiler; *linenum* is the line number on which the error occurred. If a line number and file name are generated before the colon in the last line, they are associated with your source code.

cc: const object *identifier* is not initialized

Message Name: `const_not_init`

Identifiers that are qualified by `const` must be initialized when they are declared because they cannot be assigned a value after they are declared.

Example:

```
/* compile with cc -d const_not_init=w file.c */
main()
{
    const int x;
}
```

produces the message.

`cc: constant in conditional context`

Message Name: `const_condition`

This message indicates that an `if` expression or a `switch` expression results in a constant value.

Example:

```
/* check with lint -d const_condition=w file.c */
main()
{
    int x;

    if( 4 ) /*null*/;

    switch( 4 ) /*null*/;
}
```

Conditional statements that depend on constant expressions can be simplified. The presence of these conditionals may indicate a logic error; alternatively, you may want to code these conditionals using preprocessor directives.

`cc: continue statement not in do, for, or while statement.`

`continue` statements can occur only within a `do`, `for`, or `while` statement. The following code

```
main()
{
    continue;
    return(0);
}
```

is an example of this type of error. Correct this error by deleting the `continue` statement.

`cc: conversion from unsigned value to signed value of the same size may cause change of sign`

Message Name: `cvt_changes_sign`

This message occurs when an unsigned integer is assigned to a signed integer of the same size.

Example:

```
/* check with lint -d cvt_changes_sign=w file.c */
main()
{
    unsigned h;
    signed j;
    j = h;
}
```

This assignment may result in a large positive number being converted into a negative number. Correct this error by changing the target integral type to an unsigned type.

This message may be useful in tracking down bugs that occur because of unintended sign changes. However, due to type conversion rules, the output of this diagnostic message may be rather large.

`cc: conversion to character type truncates`

Message Name: `char_cvt_truncates`

This message indicates that a **char** data type is the target of an integer that requires more storage than a **char** data type. This may result in a truncation of the original value.

Example:

```
/* check with lint -d char_cvt_truncates=w file.c */
char x;
unsigned int y=500;
x = y;
```

Correct this error by changing the target data type to a larger integral type.

This message may be useful in tracking down bugs that occur because of unintended truncation. However, due to type conversion rules, the output of this diagnostic message may be rather large.

`cc: conversion to integer type truncates`

Message Name: `int_cvt_truncates`

This message indicates that your program has a `long long` variable that is being converted to a variable of type `int`.

Example:

```
/* check with lint -d int_cvt_truncates=w file.c */
main()
{
    long long x = 1;
    int y;

    y = x;
}
```

This type of conversion has the potential for converting a large positive number into a negative number. Correct this error by increasing the size of the target integer.

This message may be useful in tracking down bugs that occur because of unintended truncation. However, due to type conversion rules, the output of this diagnostic message may be rather large.

`cc: conversion to short integer type truncates`

Message Name: `short_cvt_truncates`

This message indicates that an integer that requires more bytes for its representation than a `short` integer is being assigned to a `short` integer.

Example:

```
/* check with lint -d short_cvt_truncates=w file.c */
main()
{
    long h = 1;
    int i = 1;
    short j;

    j = i;
    j = h;
}
```

This type of conversion has the potential for converting a large positive number into a negative number. Correct this error by increasing the size of the target integer.

This message may be useful in tracking down bugs that occur because of unintended truncation. However, due to type conversion rules, the output of this diagnostic message may be rather large.

`cc: conversion to unsigned is undefined for negative values`

Message Name: **`cvt_to_unsigned`**

This message occurs when a signed integral type is assigned to an unsigned integral type. The unsigned integral type can be the same size or larger than the signed integral type.

Example:

```
/* check with lint -d cvt_to_unsigned=w file.c */
main()
{
     signed src;
     unsigned targ1;
     unsigned long long targ2;
     targ1 = src;
     targ2 = src;
}
```

An assignment of this type converts a negative number to a positive number. Correct this error by changing the unsigned target integral type to a signed integral type.

This message may be useful in tracking down bugs that occur because of unintended sign changes. However, due to type conversion rules, the output of this diagnostic message may be rather large.

`cc: Could not close CXdb data file` *'name'*.

This message indicates that the compiler could not close the *name* CXdb data file. Check your file permissions and disk space.

`cc: Could not perform` *'file-op'* `operation on CXdb data file` *'name'*.

This messages indicates that the compiler could not perform a write or fseek operation on the *name* CXdb data file. Check to make sure that you have sufficient disk space and that your data files are not corrupted.

`cc: CPU time limit exceeded`

This message occurs when the compiler exceeds the maximum amount of CPU time permitted on your CONVEX system. To increase the amount of time permitted, use the `-tl` *time* option, where *time* is the new time limit in minutes. The current CPU time limit for your system can be determined by entering the `limit` command. For further information consult the setrlimit(2) man page.

`cc: CXdb data file directory 'directory' is a plain file.`

This message occurs when the compiler is unable to open the *directory* subdirectory as a directory. One cause of this is that *directory* exists as a file instead of a directory. Correct this by renaming the *directory* file, and then recompile your program.

`cc: decimal integer must range between 1 and 32767`

This message indicates that the value of the line number argument of `#line` is not between 1 and 32767, inclusive.

Example:

```
# line 0
# line 40000
```

Correct this error by using the correct integer range.

`cc: declaration contains two storage class keywords`

This message indicates that a declaration specified more than one storage class:

```
static extern f( void );

main()
{
    register static int x = 5;
}
```

There are five storage classes: `extern`, `static`, `auto`, `register`, and `typedef`. Use of two or more of these in one declarations is contradictory.

`cc: declaration of` *identifier* `conflicts with compiler support routine`

This message indicates that your program contains an identifier that the compiler uses internally to implement features such as structure assignment.

Example:

```
int gen$bcopy;

main()
{
    struct {
        int x,y,z;
    } a,b;
    f( &a, &b );
    a = b;
}
```

Correct this error by renaming the identifier to an identifier not used internally. The compiler names that are used internally all contain the "$" character, so avoid using this character.

`cc: declared formal parameter '`*param*`' is missing.`

This message occurs in the following situation:

```
f()
int x;
{}
```

This is a function definition; the parameter is declared in the old C style. In this case, **x** does not exist in the function parameter list. The correct usage is:

```
f(x)
int x;
{}
```

cc: *'variable'* declared 'extern' within function may not be initialized

Variables declared with the `extern` storage class cannot be initialized when the declaration appears in a function definition. The following code generates this error message.

Example:

```
int main(){
    extern int var = 4;
}
```

The `extern` variables (at file scope) can be initialized only in the ANSI C modes.

cc: *directive name* directive ignored - loop or inner loop has exit

This message indicates that a loop to which you applied the *directive name* directive or pragma, contains a non-sequential exit. Statements such as `return`, `break`, `goto` or a function call may cause a non-sequential exit from a loop.

Example:

```
#include <stdio.h>
main()
{
    int i,j,sum=0;
    int a[100];

    #pragma _CNX force_vector
    for(i=0; i<100; i++) {
        if( i == 20)
            break;
        sum += a[i];
    }
end:
    printf("sum = %d\n", sum );
}
```

Correct this error by recoding the loop so that it does not depend on `return`, `break`, `goto`, or a function call to exit.

```
cc: directive name directive ignored - switch statement
```

This message indicates that a loop, to which you applied the *directive name* directive or pragma, contains a **switch** statement. For example:

```
#include <stdio.h>
int main()
{
     int i,sum=0;
     int a[100];

     #pragma _CNX force_vector
     for(i=0; i<100; i++)
         switch( i ) {
              case 5: break;
              default: sum += a[i];
         }
     printf("sum = %d\n", sum );
}
```

The compiler cannot vectorize loops that contain **switch** statements. To correct this situation, consider replacing the **switch** statement with an **if** statement. The above example can be converted to:

```
#include <stdio.h>
int main()
{
     int i,sum=0;
     int a[100];

     #pragma _CNX force_vector
     for(i=0; i<100; i++)
         if( i != 5 )
              sum += a[i];
     printf("sum = %d\n", sum );
}
```

switch statements cannot always be converted to **if** statements.

```
cc: directives specified for loop are inconsistent - some ignored
```

This message occurs when you use pragmas or directives that are incompatible with each other. Refer to Table B-1, "Restrictions on Pragma Use," for a list of incompatible pragmas and directives.

`cc: division by zero is possible at runtime.`

This message indicates that the compiler has encountered a possible divide by zero operation when folding constants during optimization.

Example:

```
#include <stdio.h>
main()
{
    int i = -1;
    int j = 0;

    if( i )
        i = i/j;
    printf("i = %d\n", i );
}
```

This message only occurs at optimization level `-O1` or higher. To correct this error you may have to trace back from the statement to determine which variable is causing the problem. In this example, initialization of `j` to zero causes the divide by zero error message when `j` is an operand of the division operator.

`cc: dollar sign character occurs in identifier` *identifier*.

Message Name: `dollar_names`

Dollar sign characters ($ are not permitted in identifiers in ANSI C source code, but they are allowed as a CONVEX extension. This message indicates a possible nonportable use of the dollar sign character.

Example:

```
/* compile with cc -d dollar_names=e file.c */
int heavy$;
```

This code produces this message.

`cc: duplicate case label.`

No duplicate **case** labels in a **switch** statement are permitted.

The following lines of code generate this error message:

```
#define A Z
#define B Z

switch( variable ){
    case A:
    case B: ;
}
```

The preprocessor converts the two macros **A** and B to the same **case** label, causing the error message to be displayed.

`cc: duplicate definition: label already defined.`

The scope of a label is an entire function. This message indicates that a label has been defined twice in one function.

`cc: embedded begin_tasks directives disallowed`

The following code fragment is illegal:

```
#pragma _CNX begin_tasks
task1();

#pragma _CNX next_task, begin_tasks
    taskA();

    #pragma _CNX next_task
    taskB();

#pragma _CNX end_tasks, next_task
task2();

#pragma _CNX end_tasks
;
```

Embedded **begin_tasks** pragmas are not necessary because the embedded tasks are inherently parallel with the outer level of tasks. For example, **taskA** in the previous code fragment is independent of **task1**, so it isn't necessary to embed it.

`cc: end_tasks directive with no begin_tasks ignored`

The **end_tasks** pragma must be preceded by a **begin_tasks** pragma. The **end_tasks** pragma is ignored if this is not the case.

```
cc: enum constant must be in range of type 'int'
```

The data representation for the `enum` data type is the same as that for the `int` data type. Consequently, an `enum` constant must not exceed the range of an `int` data type.

cc: enum tag *tag_name* may not be used before declaring its members

This is caused by using an incomplete `enum` type (for example, `enum x;`) before defining its members.

Example:

```
enum some_e x;
```

This code produces this message if the members of `some_e` have not yet been declared.

cc: error *'diagnostic name'* can not be changed to warning message or suppressed.

The indicated message cannot be either converted to a warning message, or it cannot be suppressed. For an example on the use of the `-d` compiler option, refer to the Section "Compiler Diagnostic Options," at the beginning of this appendix.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
cc: error reading lint input file.
```

`lint` creates an intermediate file when it processes your C source files. When `lint` is unable to read the intermediate file that it creates, it generates this message. The intermediate file could have been corrupted by another system process. Contact your system manager when this error occurs.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
cc: error writing lint intermediate file.
```

This message indicates that `lint` was able to open the intermediate file, but encountered an error when writing to it. The intermediate file could have been corrupted by another system process. Contact your system manager when this error occurs.

`cc: Escape sequence is out of range for character`

Message Name: `escape_range_sequence`

This message occurs when an integer constant value exceeds 0xff. For example:

```
/* compile with cc -d escape_range_sequence=e file.c */
int c = '\xff1';
```

This code produces this error message.

`cc: expression has no side effect and value is not used`

Message Name: `null_effect_expression`

This message occurs when math functions declared in the math.h include file return a value that is not used.

Example:

```
/* compile with cc -d null_effect_expression=e file.c */
#include <math.h>

main()
{
    sin(0.0);
}
```

`cc: extern '`*identifier*`' is hidden when declared by line` *line-num* `of` *file*

Message Name: `hidden_extern`

This message is only generated with source files compiled in the backward-compatible mode. It indicates that a declaration using `extern` inside a function declares an identifier that is not visible where it is declared.

Example:

```
/* compile with cc -pcc -d hidden_extern=w file.c \
    other.c */
extern int print_extern();
main()
{
    int f=10;
    {
        extern int f;
        printf("f = %d\n", f );
        f = 3;
        print_extern();
        printf("f = %d\n", f );
    }
    printf("f = %d\n", f );
}

/* other.c */
int f = 6;
int print_extern()
{
    printf("f in external function = %d\n", f );
}
```

The two source files in the example produce the following output when compiled in the backward-compatible mode:

```
f = 10
f in external function = 6
f = 3
f = 3
```

In the backward-compatible mode, all `extern` declarations have file scope, no matter where the declaration occurs. Consequently, the f variable that is initialized to 10 obscures all references to the external variable f.

This message usually identifies a situation where a program refers to a different object than was intended. To correct this error, change the name of the local identifier; in the example, change the name of the variable f that is initialized to 10.

`cc: extra tokens after` *name* `directive, ignored`

Message Name: `pp_extra`

This message indicates that the preprocessor found extra tokens in a line that contains a preprocessor directive. Possible causes include:

- `#undef` has more than one identifier when compiling in an ANSI C mode.
  `# undef x extra`
- The second argument of `#line` is not text enclosed in double quotes.
  `# line 2 extra`
- The `#line` preprocessor directive contains extra arguments following a legally formed second argument.
  `# line 3 "text" extra`
- `#if` or `#elif` has more than one argument expression.
  `# if x==y a == b`
- `#ifdef` or `#ifndef` has more than one argument expression when compiling in an ANSI C mode.
  `# ifndef d == g h`
- The `#include` preprocessor directive has additional arguments following `#file#` or `<file>`.
  `# include <stdio.h> extra`
- The `#endif` preprocessing directive has any arguments when compiling in an ANSI C mode.
  `# endif comment`

Correct these errors by correctly using the preprocessor directives. The corrections for each of the examples above are:

- `# undef x`
- `# line 2`
- `# line 3 "text"`
- `# if x==y`
- `# ifndef d`
- `# include <stdio.h>`
- `# endif /* comment */`

`cc: field size too large for field` *'member name'*

The number of bits in a bit field cannot exceed the number of bits available in the data type used to declare the bit field.

Example:

```
unsigned field_1 : 41;
```

The maximum number of bits that can be allocated to `field_1` is 32 because the `unsigned` data type has a maximum of 32 bits.

```
cc: File size limit exceeded
```

This message occurs when a file that the compiler creates exceeds the maximum file size permitted on your system. For further information consult the man page for setrlimit(2). The current file size limit for your system can be determined by entering the `limit` command.

```
cc: a floating point exception may occur here at runtime.
```

This message indicates that your code may cause a floating-point exception when it is executed. The following code is one example of this:

```
#include <math.h>
#include <stdio.h>
main()
{
     float y;
     int i = -1;

     if( abs(i) )
         y = sqrt(i);
     else
         y = sqrt(i);
     printf("y = %f\n", y );
}
```

If you compile this code at optimization level `-no` or `-O0`, the possible floating-point exception is not found: sqrt(-1). When you run the executable generated by this code, it prints out:

```
y = 1.000000
```

No errors are indicated. However, if you compile this code at optimization level `-O1` or higher, the error message is generated because the compiler has folded `-1` into all occurrences of i, and then computed `sqrt(-1)` in an attempt to reduce the number of computations the compiler must make at runtime.

To correct the error, you may have to trace your code to see which values the compiler has folded; this enables you to determine what parts of your code may generate floating-point exceptions. Refer to Chapter 9, "CONVEX C Intrinsics," for information on the relationship between intrinsic functions and floating-point exceptions.

`cc: floating point literal` *`value`* `contains suffix`

Message Name: **`float_suffix`**

This diagnostic option reports use of floating-point suffixes: f, F, l, L, d, and D. If this option is an error, these suffixes are considered syntax errors, and a syntax error message is generated; if this option is a warning, this message is generated unless compiled in the strict mode with the d or D, suffixes which result in a syntax error.

Example:

```
/* check with lint -d float_suffix=w file.c */
float real = 4.5d;
```

This example produces the message, but:

```
/* compile with cc -d float_suffix=w -str file.c */
float real = 4.5d;
```

results in a syntax error.

`cc: formal macro argument list is incomplete`

This message indicates that the formal argument list of a macro definition is not terminated by a right parenthesis, ')'.

Example:

```
# define w(r,t,s
```

Correct this error by inserting a right parenthesis at the end of the formal argument list.

`cc: formal '`*`parameter`*`' may not be initialized`

Formal function parameters cannot be initialized.

Example:

```
int f(x)
int x = 3;
{
    return (1);
}
```

Correct this error by removing the initializer—parameters are always given an initial value in the function call.

`cc: formal 'parameter' may not have function type`

Message Name: `function_parameter`

Functions cannot be parameters of a function. The following is illegal:

```
/* compile with cc -d function_parameter=e file.c */
int func(void);
int func( int func( void ) );
```

When this diagnostic is turned off or converted to a warning, the compiler interprets the declaration as being a pointer to a function instead of passing the function itself.

To legally pass a function, declare it as a pointer. The correct code is:

```
int func( void );
int func( int (*func)(void) );
```

`cc: formal parameter` *parameter number* `of function` *function name* `has no name`

All parameters in a function definition must have a name.

Example:

```
int func(int param1, int )
{
}
```

In this example, the second parameter does not have a name. Correct this error by inserting an identifier for the missing function parameter. Parameter names are not required in function prototypes.

Example:

```
int func( int, int );
```

`cc: formal 'parameter' was not explicitly declared`

Message Name: `no_arg_type`

This message indicates that a formal argument is declared without a data type.

Example:

```
/* compile with cc -d no_arg_type=e file.c */
void f(x)
{
    x = 4;
}
```

In such cases the parameter implicitly has type `int`.

`cc: function has incomplete return type`

It is illegal to call a function that has an incomplete return type. Incomplete return types do not provide enough information to enable a compiler to determine the size of the data returned.

Example:

```
int main(){
    struct unknown_s func(void);
    (void) func();
}
```

In this example, `unknown_s` is a `struct` of unknown size. While it is illegal to call a function that has an incomplete return type, it is not illegal to declare one.

`cc: function name has return(e); and return;`

This message indicates that a function has a `return` statement that contains an expression and a `return` statement that does not return an expression.

Example:

```
int far()
{
    return (1);
}

int main()
{
    if( far() )
        return (1);
    return;
}
```

The compiler detects the two different syntaxes of the `return` statement in the `main` function. Correct this error by using the same `return` syntax consistently.

`cc: function` *function name* `is declared static but never defined`

The scope of a `static` function is limited to the functions in the source file in which it is defined. This message indicates that a source file contains a `static` function declaration and a call to that function, but does not contain the definition of the `static` function. Correct the error by defining the function in the file or converting the function storage class to `extern` and defining the function in another file.

`cc: function` *identifier* `may not be declared` *storage_class* `in block scope`

ANSI C does not permit functions declared in block scope with the `auto`, `static`, or `register` storage classes.

Example:

```
void func()
{
    auto some_f();
}
```

This code produces this message.

`cc: function` *function name* `may not be initialized`

Functions cannot be initialized. However, it is possible to initialize a pointer to a function.

`cc: function returning array illegal.`

A function cannot return an array. For example,

```
int func( void )[];
```

is an illegal declaration of a function that returns an array of integers. However, functions can return the address of an array:

```
int ( *func( void ) )[];
```

returns a pointer to an array of integers.

```
cc: function returning function illegal.
```

An example that generates this message is:

```
int (func(void))(void);
```

This prototype declares a function that returns a function that returns an `int`, which is illegal.

The correct method is to return a pointer to a function:

```
int (*func(void))(void);
```

In these examples, function **func** and the function it returns have no parameters.

```
cc: function 'f' declared with prototype may not have old-style
    formal declarations
```

This message indicates that the parameters of function *f* are declared with the prototype form and with the old style parameter declaration, as in:

```
int f(int x)
int x;
{
    return (1);
}
```

These two styles of parameter declarations cannot be mixed. To correct the error, delete the old-style parameter declarations.

```
cc: global declaration has no type or storage class specifier
```

This message indicates that a declaration in file scope did not specify a data type or a storage class. This type of declaration is permitted in the backward-compatible mode of the compiler, but ANSI C prohibits it.

Example:

```
a, b(), *c;
main()
{
    c = &a;
    a = b( c );
}
```

In the backward-compatible mode, `a` is interpreted as an `int`, `b` as a function returning an `int`, and `c` as a pointer to an `int`.

To bring this code into compliance with the ANSI C standard, a data type or storage class must be specified on all declarations excluding functions.

`cc: greater than 255 tasking directives`

A maximum of 254 `next_task` pragmas (directives) can be used with each `begin_tasks` pragma (directive). Correct the error by dividing the pragmas (directives) into blocks of 255.

`cc:` '*identifier*' `hides declaration at line` *line-num* `of` *file*

Message Name: `hides_outer`

This message indicates that the declaration of an identifier hides a declaration of an identifier with the same name in an enclosing block or at file scope.

Example:

```
/* compile with cc -d hides_outer=w */
int a:
void f(void)
{
    int b;
    {
        int a; /* Hides declaration at file scope */
        int b; /* Hides decl. in enclosing block */
    }
}
```

This may indicate a flaw in the logic of your program. To correct this error, rename the identifier that obscures the identifier in the enclosing block or the identifier at file scope.

`cc: identifier expected`

This message occurs when a formal argument of a function-like macro is not a legal identifier, or an `#ifdef` or `#ifndef` does not have a legal identifier as an argument.

Example:

```
#define x(b,4
#ifdef 4
#endif
```

Correct these errors by using the correct preprocessor syntax.

```
cc: ignoring dependence from 'identifier' to 'identifier' can
    cause generation of incorrect object code !!
```

This message occurs when the compiler detects a possible loop-carried dependency. The loop-carried dependency may be between the two variables, or it could be caused by a function call. For more information on loop-carried dependencies and how they affect optimization, refer to the *CONVEX C Optimization Guide.*

```
cc: ignoring initializer on 'extern' declaration of 'identifier'.
```

This message occurs only in the backward-compatible mode of the compiler.

Example:

```
extern int i = 5;
```

The initialization of the variables declared with `extern` is ignored in the backward-compatible mode. There are several solutions to this:

- Remove the external storage class specifier (convert to: `int i = 5;`).
- Don't compile the program in the backward-compatible mode.
- Use the external storage class specifier, but initialize the identifier in the body of a function.

```
cc: illegal -D macro definition
```

This message is only generated when the C preprocessor is run by itself with the `cpp` command. It indicates that the name used with the `-D` option is an identifier not permitted in C. The following command line generates this error because the name following the `-D` option contains a symbol, #, that cannot appear in a C identifier:

```
cpp -Dwer#re file.c
```

Correct this error by removing the illegal symbol.

```
illegal -U undefinition, ignored
```

Message Name: `pp_badflag`

This message indicates that you attempted to undefine a macro that cannot be undefined. The macros that cannot be undefined are `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__` (in the standard and strict modes), and `defined`.

`cc: illegal character in source (ignored)`

This message occurs when a character not in the C character set is found in the source code. Several such characters are: ', `, and @.

`cc: illegal indirection.`

This message occurs when the indirection operator, *, is incorrectly applied. The operand of this operator can be:

- pointer to a function
- pointer to an object
- pointer to a type

All other applications of this operator are illegal.

This operator is also used to evaluate subscript expressions. For example, the subscript expression

expression-1 [*expression-2*]

is equivalent to

*((*expression-1*) + (*expression-2*))

Consequently, when the contents of the outermost parentheses do not evaluate to one of the three pointer types mentioned previously, the `illegal indirection` message is generated.

Example:

```
int i = 5;

i[3] = 4;
```

In this case, the operand of the * operator, which is used implicitly in the subscript expression, is `i + 3`. The expression `i + 3` is not a pointer, so an error occurs.

`cc: illegal initialization.`

Permitted initializations:

- Static variable with a constant expression
- Automatic variables with an expression
- Pointers with constant expressions that evaluate to a pointer of the correct type

All other initializations are illegal.

`cc: illegal integer constant suffix`

The following integer constant suffixes are available in all modes of the compiler:

- L or l: suffixes for `long` data types
- U or u: suffixes for `unsigned` data types

Combine L and U for unsigned long data types. The LL or ll integer suffixes are available only in the extended mode; these suffixes are used for the `long long` integral data types.

`cc: illegal macro undefinition`

This message occurs when the `#undef` preprocessor directive:

- Does not have an argument
- Has an argument that is not a legal identifier
- Has an argument that is either `__FILE__`, `__LINE__`, `__DATE__`, `__STDC__` (in standard and strict modes), or `__TIME__`

Example:

```
#undef
#undef 234
#undef __DATE__
```

Correct this error by using a legal argument of `#undef`.

`cc: illegal pointer subtraction.`

This message occurs when pointers of different types are subtracted from each other.

Example:

```
char *p;
int *q;

p -= q;
```

This message can be eliminated by declaring both identifiers as pointers to the same type.

```
cc: illegal pointer/integer combination.
```

There are several causes of this message:

- Initializing pointer data types with expressions that are not pointer types
- Initializing nonpointer data types with expressions that are pointer types
- Combining integers and pointers in illegal operations

```
cc: illegal preprocessing directive syntax, skip to end of line
```

This message indicates that:

- More parts of a preprocessor directive are required.
- Only white space is encountered after a `#define`.
- The text following `#define` does not specify an identifier.
- A `#include` preprocessor directive is not terminated by a > directive.
- The argument of the `#include` preprocessor directive could not be expanded into a legal form.

Examples:

```
# line
# define
# define 345
# define !asdf
# include < stdio.h
# include something
```

Correct errors of this sort by using legally formatted preprocessor directives. The above examples could be corrected with:

```
# line 24
# define blank
# define x345
# define asdf
# include <stdio.h>
# define something <string.h>
# include something
```

```
cc: illegal storage class for function.
```

The only storage classes that are available for function declarations are `static` and `extern`. A function that has a `static` storage class in its definition cannot be used by functions that are defined in other source files. The `extern` storage class is the default storage class for functions.

```
cc: illegal storage class for parameter declaration.
```

Only the **register** storage class keyword can be explicitly used in a parameter declaration.

```
cc: illegal struct or union combination.
```

Structures that are assigned to each other must be declared with the same structure definition.

Example

```
struct {int *r;} *p;
struct {int *r;} *q;

p = q;
```

The identifiers p and q are declared with the same definition:

```
struct {int *r;} *p, *q;

p = q;
```

No message occurs. Similarly, if a structure tag is used:

```
struct tag {int *r;};
struct tag *p;
struct tag *q;

p = q;
```

no message occurs because the same structure definition is used to declare p and q.

```
cc: illegal type combination.
```

This message occurs when two types are used to declare an identifier, as in:

```
int float x;
```

Only one data type can be used to declare an identifier.

```
cc: illegal type for bit field.
```

If a bit field is not an `int` or `unsigned int`, a warning message occurs. For example, `float x:3;` can cause this message. However, integral types, such as `char` and `long` (and `long long` in the extended mode) are permitted.

```
cc: illegal type: function type can not be 'const/volatile' qualified
```

This message indicates that a function type defined with `typedef` was qualified with a `const` or `volatile` qualifier:

```
typedef int f();
volatile f x;
```

Correct this error by removing the incorrect qualifier or embedding it in the `typedef` declaration.

```
cc: illegal type: 'const' appears twice in declarator
```

This message occurs when the `const` qualifier appears twice in the declarator of a declaration.

Example:

```
int * const const x;
```

Correct this error by deleting the duplicate qualifier. It is possible for a qualifier to appear twice in a declaration:

```
const int * const x = 5; /* const ptr to const int */
```

In this case, one qualifier occurs in the declarator, and one in the type specification of the declaration. Similarly, it is possible for two different qualifiers to appear in the declarator:

```
int * volatile const x = 5;
```

```
cc: illegal type: 'volatile' appears twice in declarator
```

This message occurs when the `volatile` qualifier appears twice in the declarator of a declaration.

Example:

```
int * volatile volatile x;
```

Correct this error by deleting the duplicate qualifier. It is possible for a qualifier to appear twice in a declaration:

```
/* volatile ptr to volatile int */
volatile int * volatile x = 5;
```

In this case, one qualifier occurs in the declarator, and one in the type specification of the declaration. Similarly, it is possible for two different qualifiers to appear in the declarator:

```
int * volatile const x = 5;
```

`cc: illegal type: 'const' appears twice in type specifier`

This message occurs when the **`const`** qualifier appears twice in a variable declaration.

Example:

```
const const int x;
```

Correct this error by deleting one of the `const` qualifiers. *See also* `cc: illegal type: 'const' appears twice in declarator`

`cc: illegal type: 'volatile' appears twice in type specifier`

This message occurs when the `volatile` qualifier appears twice in a variable declaration.

Example:

```
volatile volatile int x;
```

Correct this error by deleting one of the `volatile` qualifiers. *See also* `cc: illegal type: 'volatile' appears twice in declarator`

`cc: illegal type: 'const' on type which is already 'const'`

This message occurs when `const` is used to qualify a type defined with typedef that was already qualified by `const`.

Example:

```
typedef const int cint_t;
const cint_t x;
```

Correct this error by deleting one of the `const` qualifiers.

`cc: illegal type: 'volatile' on type which is already 'volatile'`

This message occurs when `volatile` is used to qualify a type defined with `typedef` that was already qualified by `volatile`.

Example:

```
typedef volatile int vint_t;
volatile vint_t x;
```

Correct this error by deleting one of the `volatile` qualifiers.

```
cc: illegal {.
```

This message indicates that there are redundant braces around the initializers for a bit field.

Example:

```
struct {
    int bits : 4;
} x = {{{3}}};
```

Correct this by removing the extra braces.

```
cc: illegal '..' in source, ignored
```

This message occurs when the compiler encounters a `..` in the source code.

`cc: implicit declaration "extern int` *function_name*`()" supplied`

Message Name: `implicit_decl`

This message indicates that an implicit function declaration is detected.

Example:

```
/* compile with cc -d implicit_decl=e file.c */
main()
{
    float rc = some_func();
}
```

This code produces this error message.

`cc: include file` *file* `not found`

This message indicates that the preprocessor was unable to locate *file*. When *file* is delimited by double quotes, the preprocessor looks in the following locations (in order of first place searched to last place searched):

1. Directories specified by the `-I` preprocessor option
2. Current file directory
3. System directory, /usr/include

When *file* is delimited by angle brackets, <*file*>, the preprocessor looks in the following locations:

1. Directories specified by the `-I` preprocessor option
2. System directory, /usr/include

```
cc: incorrect use of the stringizing operator #
```

Message Name: `pp_badstr`

This message indicates that the operand of the stringizing operator (#) is not one of the formal arguments in the formal argument list of the function-like macro definition.

Example:

```
/* compile with cc -d pp_badstr=w file.c */
# define x(f) #g
```

Correct this error by ensuring that the operand of the stringizing operator is an argument in the formal argument list.

```
cc: incorrect use of the tokenpasting operator ##
```

Message Name: `pp_badtp`

This message indicates that the result of combining the left and right operands of the ## macro definition operator is not:

- A legal operator
- An identifier that has been defined
- A legal number

Examples:

```
/* compile with cc -d pp_badtp=w file.c */
# define op(b,c) b##c
x op(+,-) 5;                              /* x +- 5 */
# define id(num) "id" ## num
int id = id(3);                           /* int id = id3 */
# define frac(dot,num) dot ## num
float x = 6frac(.,z);                     /* float x = 6.z */
```

To correct errors like these, it may be useful to examine the output of the preprocessor using the −E preprocessor command line option.

```
cc: integer constant expected.
```

Integer constants are required in some C contexts. One such context is the length of a bit field.

Example:

```
int y = 3;
struct { int x : y; } z;
```

This code is illegal.

The correct code is shown below:

```
struct { int x : 3; } z;
```

`cc: integer literal` *value* `contains long long suffix`

Message Name: `long_long_suffix`

The `long long` data type is a CONVEX extension. This diagnostic option detects a `long long` integer suffix (ll or LL).

Example:

```
/* check with lint -std -d long_long_suffix=e file.c */
long long int x = 4LL;
```

`cc: integer literal` *value* `contains unsigned suffix`

Message Name: `unsigned_suffix`

This message occurs when the compiler detects an integral suffix U or u.

`cc: invalid integer expression found while parsing #if:` *message*

This message indicates that a parsing error occurred in a `#if`, or `#elif` preprocessor directive. *message* provides additional information on the cause of the error.

Example:

```
#if 4.
#endif
```

This code generates the following two *messages*:

```
illegal token
syntax error
```

cc: `invalid wide character constant` *name*. `Length must be 1.`

A character constant that has an L prefix is a wide character constant. The wide character constant has a maximum length of one character.

Example:

```
char wch = L'ss';
```

This code is illegal. The extended and backward-compatible modes permit a character constant to have between one and eight characters, while the strict and conforming modes permit between one and four characters.

However, if the character constant must be a wide character constant, delete the extra characters.

cc: '*identifier*' `is not a DIRECTIVE.`

This message is displayed when the compiler does not recognize a pragma (directive). The pragmas (directives) are described in Appendix B, "Pragmas."

cc: `label` '*name*' `defined but not referenced.`

This message indicates that label *name* is not used by a `goto` statement in your program.

Example:

```
main()
{
     label:
     ;
}
```

Correct this error by removing the unused label.

cc: `label` '*label name*' `referenced but not defined.`

The indicated *label name* is used in a `goto` statement, but it is not defined anywhere in the source file as a label.

cc: `left operand of -> or . does not have a member` '*member name*'.

This message occurs when *member name* is not a member of the `struct` that declared the left operand of the -> or . operator.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
cc: Lint file does not match this version of lint
```

The `lint` program is dependent on `lint` versions of all standard libraries that accompany the C compiler. `lint` libraries contain information that `lint` uses to ensure that functions in the library are called correctly by the functions in your program. The `lint` libraries have names of the form llib-l*name*.ln. This error message indicates that the specified `lint` library was created using a previous version of the `lint` program. If you created the `lint` library using a previous version of `lint`, you must create a new library using the current `lint` program. Otherwise, inform your system manager when this error occurs.

```
cc: loop directive not textually immediately before loop.
```

Pragmas (directives) that affect loops must immediately precede that loop.

Example:

```
typedef int Mat[100][28];
int main()
{
    int i,j,n,m=100;
    Mat a,b;

    #pragma _CNX scalar
    n = 28;
    for(j=0; j<n; j++)
        for(i=0; i<m; i++)
            a[i][j] = b[i][j] + 1;
}
```

The assignment to `n` between the pragma and the loop causes the message.

```
cc: lvalue required.
```

An lvalue (locator value) is an expression that locates a data object in memory. Possible lvalues are:

- Arithmetic variables
- Pointer variables
- Enumerated variables
- Structure variables
- Union variables

Also:

- A subscript expression *array-name*[*integral-expression*] is an lvalue.
- The value resulting from a `struct/union` . or -> operator is an lvalue if the left operand is an lvalue. (func().x is not an lvalue.)
- If the operand of the indirection operator, *, points to a data object, not a function, the result of the operator is an lvalue.
- A parenthesized expression is an lvalue if its unparenthesized expression is an lvalue.

Some operators that require an lvalue are: ++, --, =, and unary &.

If one of these requirements is not met, an appropriate message is generated.

Example:

```
int b[5];

b = 5;
```

The assignment operator generates a message. Even though `b` is the left operand of an assignment operator, it is also the base address of an array which can never be modified.

Example:

```
int func(void), r(void);

main()
{
    func = r;
    return(0);
}
```

The left operand of an assignment operator cannot be modified, because the addresses of functions cannot be modified. Similarly, `const` qualified data types cannot be modified. The following code fragment contains an illegal assignment:

```
const int x = 5;    /* initialization is legal */

x = 10;
```

Because `const` qualified data types are not lvalues, the second assignment statement is illegal. Another constant data type that is not an lvalue is an enumerated constant.

```
cc: this machine does not have hardware support for IEEE.
```

This message occurs when the compiler is invoked with the `-fi` command line option on a computer that is not equipped with the IEEE support hardware.

If the machine that the program runs on is not the same as the machine used to compile the program, both machines must have hardware support for IEEE if such hardware is required by the program.

```
cc: macro actuals ended unexpectedly, inserting )
```

This message indicates that an actual argument list of a function-like macro does not have a terminating right parenthesis.

Example:

```
int j;
# define a(b,c,d) b##c##d
a(i,n,t
    j;
```

In this case, the preprocessor inserts the right parenthesis, permitting the `j` identifier to be declared as an `int`. Correct this error by inserting the right parenthesis.

cc: macro formal argument *identifier1* and *identifier2* are not distinct

Message Name: `pp_macro_arg`

This message indicates that the formal arguments of a function-like macro defined with `#define` have the same name.

Example:

```
/* compile with cc -d pp_macro_arg=w file.c */
# define x(y,y) y##y
```

Correct this error by making each of the formal parameters distinct and convert each reference to the formal parameters in the macro definition to the appropriate name.

`cc: macro` *name* `redefined with a new replacement list`

Message Name: `pp_macro_redefinition`

This message indicates that the replacement list for the *name* macro definition is not the same as in its original definition.

Example:

```
/* compile with cc -d pp_macro_redefinition=w file.c */
# define foo(x,y) x##y
# define foo(a,b) a##b
```

Replacement lists in a macro redefinition *must* be exactly the same. However, when the preprocessor encounters this error, it replaces the previous definition with the current definition.

`cc: malformed number:` *number*

This message indicates that the preprocessor has encountered an illegally formed number; these numbers have a letter embedded in them, or an illegal suffix.

Example

```
int x = 0x34g;
int y = 23145;
```

Correct this error by converting the incorrect number to a legally defined number.

`maximum number of scalars exceeded. Restructure your computations.`

This message occurs when the number of scalar variables referenced in a function exceeds the size of a compiler table. Multiple references to a variable identifier require one table entry; multiple references to the same variable using different identifiers require one entry per identifier. Currently, the maximum number of entires allowed is 8000.

`cc: Maximum optimization level available is` *level*

This message occurs if the compiler does not support the specified optimization option. There are two versions of the compiler: one that performs scalar optimizations and one that performs vector and parallel optimizations in addition to scalar optimizations.

Example:

```
cc -O2 file.c
```

This command line generates the message if the scalar version of the compiler is being used because -O1 (or -O) is the maximum level of optimization on the scalar compiler.

```
cc: misplaced #elif
```

This message indicates that a #elif preprocessor directive follows a #else preprocessor directive without an intervening #if, #ifdef, or #ifndef preprocessor directive.

Example:

```
#ifndef __FILE__
#else
#elif 1
#endif
```

This may indicate a logic error, or an extra #elif preprocessor directive.

```
cc: misplaced #else
```

This message indicates that an #else preprocessor directive follows another #else preprocessor directive without an intervening #if, #ifdef, or #ifndef preprocessor directive.

Example:

```
#ifndef __FILE__
#else
#else
#endif
```

This may indicate a logic error, or an extra #else preprocessor directive.

cc: misplaced comma in formal macro argument list

This message indicates that the preprocessor found a comma or a right parenthesis when an identifier was expected in the formal argument list of a function-like macro definition.

Example:

```
# define a(,b)
# define c(d,)
```

Correct this error by removing the unnecessary comma or by inserting another identifier.

cc: misplaced lint directive.

Message Name: **misplaced_lint_directive**

This message indicates that a **lint** directive occurs in the wrong context. Possible causes of this message are:

- ARGSUSED directive does not occur in file scope.
- LINTLIBRARY directive does not occur in file scope.
- NOTREACHED directive occurs in file scope.
- VARARGS directive does not occur in file scope.

Turn off this message by checking your source code with **lint -d misplaced_lint_directive file.c**.

cc: missing #endif

This message indicates that an **#if**, **#elif**, or **else** does not have a terminating **#endif**. The following example demonstrates this problem:

```
main()
{
    int x;
#if y
}
```

Correct this error by including the **#endif** directive in the correct location.

```
cc: missing component for -k in preprocessing input file name
```

Message Name: `pp_badkfile`

This message is only generated when the C preprocessor is run by itself with the cpp command. It indicates that the -k file is specified on the command line without an input file. In this case, the preprocessor is expecting input from stdin. However, the -k option does not process stdin input.

```
cc: negative constant converted to unsigned type
```

Message Name: `negative_to_uns`

This message indicates that a negative constant is assigned to an unsigned type.

Example:

```
/* check with lint -d negative_to_uns=w file.c */
unsigned x;
x = -1;
```

The conversion of a negative constant to an unsigned type results in a very large positive number. If this is not the intention of the assignment, change the type of the identifier.

cc: `negative field size for field` '*member name*'.

The integer used to specify the number of bits in a bit field must be a positive integer that is less than the number of bits required to represent the type of the field.

```
cc: nested macro invocation is missing complete argument list
```

This message indicates that you have some nested macros which do not have a complete argument list.

Example:

```
#define a b(3
#define b(i) (i+1)
#define c(x) x
c(a)
```

Correct this error by ensuring that your nested macros have complete argument lists.

```
cc: Newline in string or character constant (inserting close quote)
```

Newline characters cannot be embedded in string constants or character constants.

Example:

```
char c[] = "asdf
;lkj";
```

The compiler tries to recover from the error by inserting a double quote before the newline character (after "f"). If a newline character must be embedded in a string constant or character constant, use the newline character constant, "\n". For example, one correct version of the previous example is:

```
char c[] = "asdf\n;lkj";
```

```
cc: next_task directive with no begin_tasks ignored
```

The **next_task** pragma must be preceded by a **begin_tasks** pragma. The **next_task** pragma is ignored if this is not the case.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
NO AVAILABLE MEMORY
```

The compiler has insufficient memory to compile the source file. Contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Refer to the contact(1) man page to learn how to contact the TAC.

```
cc: no comma separating formal arguments to macro
```

This message indicates that the formal arguments of a function-like macro definition are not separated by commas.

Example:

```
# define f(x y) x##y
```

Correct the error by inserting commas in appropriate locations.

```
cc: no digits in exponent of floating point constant
```

This message occurs when the exponent part of a floating-point constant does not contain any digits.

Example:

```
float value = 75.E;
```

Correct this error by including digits in the exponent part of a floating-point constant.

```
cc: no digits in hexadecimal constant
```

A number beginning with `0x` or `0X` is a hexadecimal constant. If no hexadecimal digits follow either of these prefixes, this message occurs.

Example:

```
int value = 0x;
```

Correct this error by following the hexadecimal prefix with hexadecimal digits.

```
cc: no matching #if
```

This message indicates that a `#endif` preprocessor directive occurred without a preceding `#if`, `#ifdef`, or `#ifndef`.

```
cc: no members of this type declared
```

This message indicates that the type of a `struct` or `union` member was declared, but no identifier was included in the declaration.

Example:

```
struct s { int; };
```

Correct this error by including an identifier in the declaration.

cc: `non-standard syntax at or before` '*character*'

Message Name: `strict_syntax`

This message is caused by not placing a semicolon after the last member in a `struct` or `union` declaration or by placing a comma after the last enumeration constant in an `enum` declaration list. These syntaxes are supported for backward-compatibility and are not portable.

`cc: Nothing was declared by this declaration`

Message Name: **nothing_declared**

This message is caused by empty declarations such as:

```
/* compile with cc -d nothing_declared=w file.c */
int ;  /* or */

struct { int x; };
```

`cc: NULL at end of string initializer 'characters' truncated`

This message occurs when the number of characters in the string initializer of an array is equal to the number of elements in the array.

Example:

```
char arrx[5] = "12345";
```

The message indicates that there is not enough room in the array for the string to be terminated by a NULL character. You should increase the dimension of the array to allow the NULL character because it is expected by most string handling functions. *See also* **cc: too many initializers**.

`cc: null dimension.`

This message occurs when an array has no specified size for one of its dimensions.

Example:

```
int a[5][];
```

All indexes must be included in a declaration so that the compiler can allocate sufficient space for the array.

`cc: Octal constant contains digit 8 or 9`

Old versions of C permitted the digits 8 and 9 to be used as octal digits. ANSI C does not permit this.

Example:

```
#include <stdio.h>

main()
{
    int x = 09;

    printf("x = %o\n", x);
}
```

The output of this program is `x = 11` because the digit 9 is an octal 11 and the digit 8 is an octal 10.

`cc: old form assignment operator used.`

Some non-ANSI C compilers permitted assignment operators of the form =*op* to be used, where *op* could be one of `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, or `|`. The following example demonstrates an ambiguous situation:

```
#include <stdio.h>

main()
{
    int x = 2;

    x =- 3;
    (void)printf("x = %d\n", x );
}
```

When this program is executed, `x = -3` is displayed. If you compile with `cc -d old_form_assign file.c`, `x = -1` is displayed.

`cc: old style directive, use #pragma`

Message Name: `pp_old_dir`

This message indicates that a directive of the form `/*$dir` *directive* `*/` is used. Replace directives that use this form with the pragma syntax. Refer to Appendix B, "Pragmas," for information on the new format of compiler directives. co

Example:

```
/* check with cc -d pp_old_dir=w file.c */
main()
{
     int x;
     /* $dir no_side_effects (func) */
     x = func();
}
```

This message can be removed by using the pragma syntax for a directive, as in:

```
main()
{
     int x;
     #pragma _CNX no_side_effects( func )
     x = func();
}
```

You can use pragmas in all compatibility modes.

`cc: operand of ! must have scalar type`

The operand of the logical operator `!` must be an expression that evaluates to an integral value, a floating-point number, or a pointer.

`cc: operand '`*identifier*`' of no_side_effects is not a function identifier.`

This message occurs when the *identifier* is not a function name. For example:

```
int func(void);

void main()
{
     int bad;
     #pragma _CNX no_side_effects (bad)
     z = func();
}
```

Correct this error by using the correct function name as an argument of the `no_side_effects` pragma.

`cc: The operand of unary minus must have arithmetic type`

The minus operator that takes one operand is called a unary minus operator; it reverses the sign of its operand. Its operand must be one of the following types:

- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`, `signed short`
- `int`, `unsigned int`, `signed int`
- `long`, `unsigned long`, `signed long`
- `long long`, `unsigned long long`, `signed long long` (this is a CONVEX extension)
- Enumerated types
- `float`, `double`, `long double`
- `long float` (a CONVEX extension available only in the backward-compatible mode)
- A qualified form of one of these types

`cc: The operand of unary plus must have arithmetic type`

The plus operator that takes one operand is called a unary plus operator; it does not change the sign of its operand. Its operand must be one of the following types:

- `char`, `unsigned char`, `signed char`
- `short`, `unsigned short`, `signed short`
- `int`, `unsigned int`, `signed int`
- `long`, `unsigned long`, `signed long`
- `long long`, `unsigned long long`, `signed long long` (this is a CONVEX extension)
- Enumerated types
- `float`, `double`, `long double`
- `long float` (a CONVEX extension available only in the backward-compatible mode)
- A qualified form of one of these types

`cc: operand to function call operator () does not identify a function.`

The operator `()` has two operands: an expression that precedes the left parenthesis and an argument list delimited by the left and right parentheses. The expression must result in a pointer to a function, a pointer to a pointer to a function, etc. Any other expression results in this message.

`cc: operands of` *operand* `have incompatible types.`

Each operator in C has requirements on what can be used as its operands. For example, operands of the – operator can both be arithmetic, can both be of the same pointer type, or the left operand can be a pointer only if the right pointer is an integer. The combination of any other data types results in the above diagnostic message. Similar situations occur for the remaining operators.

`cc: operands of` *relational operator* `must be pointers to either compatible object types or compatible incomplete types.`

This message occurs when two incompatible data type pointers are compared.

Example:

```
void func()
{
    int *p;
    char *q;

    if(p < q);
}
```

Compatible pointers include:

- Pointers to compatible data types
- Pointers to arrays of unknown size
- Pointers to structures of unknown size
- Pointers to `void`

All other pointer comparisons, including pointers to function types, are not allowed.

Compatible data types include:

- Types that have the same data type
- Structure types that have the same number of members, the same member names, and compatible member types
- Union types that have the same number of members, the same member names, and compatible member types
- Enumerated types that have the same number of members, the same member names, and compatible member types
- Declarations that refer to the same object or function
- Enumerated types are compatible with integral types
- Qualified types that have the identically qualified version of a compatible type

`cc: operands of` *operand* `point to incompatible types.`

This message indicates that the two operands of *operand* are pointers to incompatible types.

Example:

```
main()
{
    int *x;
    float *y;

    x -= y; /* or */
    x = y;
}
```

Correct this situation by making the operands of *operator* compatible. *See also* `operands of` *relational operator* `must be pointers to either compatible object types or compatible incomplete types.`

```
cc: Optimization level lowered to -no in function calling setjmp
cc: Optimization level lowered to -no in function calling _setjmp
cc: Optimization level lowered to -no in function calling sigsetjmp
```

When the `setjmp` family of functions is called and the return value is ignored, the optimization level is lowered to `-no`. This allows the following code to work:

```
#include <setjmp.h>
main()
{
    volatile int flag;
    jmp_buf jb;

    flag = 0;
    (void)setjmp(jb);
    flag++;
    if ( flag == 1 ) {
        printf("I didn't get here from longjmp\n");
        longjmp(jb,1);
    } else {
        printf("I did do a longjmp!\n");
    }
    exit(0);
}
```

At higher optimization levels it appears that the `if` clause is executed unconditionally. If this message occurs, rewrite the code using the more traditional style as shown in this example:

```
#include <setjmp.h>
main()
{
    jmp_buf jb;

    if (setjmp(jb) == 0){
        printf("I didn't get here from longjmp\n");
        longjmp(jb,1);
    } else {
        printf("I did do a longjmp!\n");
    }
    exit(0);
}
```

The compiler correctly compiles this code at each optimization level.

`cc:` *'-string'* `option not recognized`

This message occurs when the compiler does not recognize a command line option passed to it. This could be an option such as:

```
cc -unknown
```

There is no `-unknown` option. This message also occurs when an option argument is not recognized.

Example:

```
cc -or function
```

The `-or` command line option does not have a `function` argument.

`cc: order of evaluation undefined for` *'expression'*

Message Name: `eval_order`

This message indicates that *expression* has an undefined order of evaluation.

The following source code has two examples that generate this message:

```
/* check with lint -d eval_order=w file.c */
extern int f(int a, int b);

main()
{
    int x=5, y=6;

    y=7, y=1;
    x = x++ + y; /* example 1 */
    return( f( x, x=4 ) ); /* example 2 */
}
```

Order of evaluation is undefined whenever a variable's value changes more than once in a statement. This is demonstrated in example 1. There are two exceptions: arguments of functions and operands of the comma operator. A function argument assigned a value and passed in two or more argument positions, as in example 2, generate this message. However, the comma operator permits one variable to be assigned multiple values in one statement, as shown in the code above.

Correct this error by splitting your statement into several statements. For example, the above code could be rewritten as:

```
extern int f(int a, int b);

main()
{
    int x=5, y=6, tmp;

    y=7, y=1;
    x = x + y; /* example 1 */
    x++;
    tmp = 4;
    return( f( x, tmp ) ); /* example 2 */
}
```

`cc: pointer cast may introduce inefficient alignment`

Message Name: `pointer_alignment_efficiency`

This message indicates that the evaluation of the expression on the indicated line may result in decreased memory efficiency. The compiler achieves maximum efficiency when data are given an alignment equal to the size of the item in bytes. For example, short integers are aligned on 2-byte boundaries, and long integers are aligned on 4-byte boundaries. However, `long long` integers need only be aligned on 4-byte boundaries.

In the following example, a pointer to a short integer is cast into a pointer to a long integer:

```
/* check with lint -d pointer_alignment_efficiency=e \
    file.c */
int f()
{
    short *a;
    void *x;

    x = (long *)a;
}
```

The compiler detects that a long integer might be aligned on a 2-byte boundary. One way to correct this error is to change **a** to a pointer to `long`.

`cc: Pointer operands of` *operator* `must reference object types`

This message occurs when pointer arithmetic is performed on pointers to `void`, pointers to incomplete types, or pointers to functions. For example:

```
void func()
{
    void *p;

    p += 2;
}
```

This is illegal because the objects pointed to have an unknown size.

`cc: pointer to function type involved in cast`

Message Name: `function_pointer_cast`

This message indicates that a pointer to an object is being cast to a function pointer.

Example:

```
/* compile with cc -d function_pointer_cast=e file.c */
int (*g)();
char *h;

main()
{
    g = (int (*)()) h;
}
```

Correct this error by either changing the type of h to a pointer to a function, or removing the incorrect assignment.

`cc: pointer to function used in arithmetic`

It is illegal to perform arithmetic operations on function pointers.

Example:

```
int noid(void){
    return(3);
}

int (*func)(void);
main()
{
    func = noid;
    func += 2;
    (*func)();
}
```

This code produces an executable program that may or may not run. Redesign this code so that no pointer arithmetic operations are performed.

`cc: pointer to void type involved in cast`

Message Name: `void_pointer_cast`

This message indicates that an object of type pointer to `void` is involved in a cast operation. The following sample code shows three examples that cause this error message:

```
/* check with lint -d void_pointer_cast=e file.c */
main()
{
    int *i, *j;
    void *v;

    i = (int *) v;
    v = (void *) j;
    i = (void *) j;
}
```

Any cast operation that assigns its result to a `void *` object, or casts a `void *` object causes this message. Also, if the type name of the cast is `void *`, the message is generated. Correct this error by using a different pointer type.

```
cc: positive decimal integer not found after #line
```

This message indicates that the first argument of the `#line` preprocessor directive is not a positive decimal integer.

Example:

```
# line @
# line "asdf"
# line 0xff
# line -1
```

The correct syntax of the `#line` preprocessor directive is:

```
# line decimal_integer ["filename"]
```

where *decimal_integer* is the new line number and the optional *filename* is the new file name.

```
cc: Possibly nested comments - /* seen in comment
```

This message occurs when the characters /* are found in a comment. This indicates a possible error, because comments cannot be embedded. Conditional preprocessor directives are recommended for use when blocks of code containing comments must be temporarily disabled.

```
cc: pragma not recognized by CONVEX
```

Message Name: `pp_unrecognized_pragma`

This message indicates that the word that follows the **pragma** directive on the indicated line is not _CNX.

Example:

```
/* check with cc -d pp_unrecognized_pragma=w file.c */
main()
{
    #pragma no_side_effects(func)
}
```

All CONVEX pragmas must be preceded by _CNX.

```
cc: preprocessing #error directive: text
```

This message does not indicate an error. It is displayed whenever the `#error` preprocessor directive is encountered in a file.

`preprocessing input file` *directory/file* `not found`

This message indicates that the preprocessor was unable to find *file* that was specified on the command line.

`cc: program contains loop to self.`

This message indicates that your program contains code that may result in an infinite loop. The following program contains some code that prevents the program from halting:

```
main()
{
    label:
    ;
    if(1 == 0)
        return;
    goto label;
}
```

Correct this error by re-coding your program so that it does not contain an infinite loop. This error is detected only at optimization level -O1 and above.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
```
source file: `This program is too complex - the parser's stack overflowed.`

The complexity of the source file caused the storage in the compiler's parser stack to be exceeded. Modularization of source files and functions reduces the chance of this error occurring. Contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Refer to the contact(1) man page to learn how to contact the TAC.

```
>>>>>   C O M P I L E R   E R R O R   <<<<<
>>>>> See your system manager for help <<<<<
This program is too complex - too many names for symbol table.
```

The storage for the compiler symbol table was exceeded. This can be caused by having too many identifiers in the source file. One solution is to split the source file into several smaller files. Contact your system manager or the CONVEX Technical Assistance Center (TAC). Refer to the contact(1) man page to learn how to contact the TAC.

cc: real constant either too large or too small.

The constant indicated by the message cannot be represented as a real constant. The minimum and maximum real constants of the `float`, `double`, `long float`, and `long double` are contained in the float.h include file.

cc: *'variable'* redeclared: appears twice in formal parameter list

This message is displayed for each duplicate of an identifier in a function formal parameter list or prototype.

Example:

```
int func(int a, int a, int a);
```

This code causes this message to be displayed twice. Correct this error by renaming the duplicate identifiers.

cc: *'function name'* redeclared: body of this function already seen

Each source file may contain only one function definition for each function.

Example:

```
int func()
{
    return(2);
}

int func()
{
    return(2);
}
```

This code generates the message because the function `func` is defined more than once. The definition need not be the same for this error to occur. Correct this error by deleting one of the function definitions.

cc: *'identifier'* redeclared: can not change storage class from extern to static after referencing it.

This message occurs when an identifier is redeclared with the `static` storage class after being declared with the `extern` storage class.

Example:

```
extern int i;

void f()
{
  i = 1;
}
static int i = 5;
```

When a reference is made to the identifier between the two storage class declarations, the error message occurs. But if no reference to the identifier is made between the declarations, the identifier is declared with the `static` storage class.

`cc:` *'variable'* `redeclared: declaration as typedef can't override previous declaration.`

An identifier that has already been defined cannot be redeclared as a `typedef` name.

Example:

```
int ident;
typedef char ident;
```

Correct this error by renaming either the `typedef` type or the first identifier.

Identifiers that are labels, tags, or members of structures and unions can have the same name as a `typedef` name because the compiler is able to distinguish between the two names based on the syntactic context.

`cc:` *'function'* `redeclared: incompatible types.`

This message occurs when the return type or argument of a function is not consistent between the function's declaration and definition. For example, after the following declaration:

```
int f(int ch);
```

either of these declarations is illegal:

```
int f(char ch);
int f(const int ch);
```

cc: *'variable'* redeclared: recursive struct or union declaration

Recursive definitions of structures are not permitted.

Example:

```
struct some_s {
    struct some_s {
        int x;
    } y;
    int z;
} w;
```

Such a declaration is meaningless because the interior **some_s** is not the same as the exterior **some_s**.

If a self-referencing structure is required, use of pointers is suggested.

Example:

```
struct some_s {
    struct some_s *sptr;
    int z;
} w;
```

This structure declares a pointer that can point to a copy of itself.

cc: *'variable'* redeclared: saw 'static' definition and then external definition.

This message occurs when a **static** function or variable is redeclared without a storage class keyword.

Example:

```
static int i;
int i;
```

Such functions or variables may be redeclared with either **static** or **extern**. Correct this error by redeclaring the variable with a storage class keyword, or delete the redeclaration. *See also* **cc**: *'variable'* **redeclared: saw external definition and then 'static' definition. cc:** *'identifier'* **redeclared: can not change storage class from extern to static after referencing it.**

cc: '*identifier*' redeclared: saw extern declaration and then static declaration; using static.

Changing *identifier* from the `extern` storage class to the `static` storage class is not defined in the ANSI C standard. For example, the semantics of the following are not defined:

```
extern int i;
static int i = 5;
```

If no reference to *identifier* is made between the declarations, the identifier is declared with the `static` storage class. This message may also occur when a function is first declared with the `no_side_effects` pragma and then declared with the static storage class:

```
#pragma _CNX no_side_effects( func )
static int func();
```

When the `no_side_effects` pragma has as an argument a function that has not been defined or declared, that function is implicitly declared as:

```
extern int func();
```

When this happens, precede the `no_side_effects` pragma with a function prototype of your affected function.

cc: '*variable*' redeclared: saw external definition and then 'static' definition.

This message occurs when a function or variable is declared with the `extern` storage class and then redeclared with the `static` storage class.

Example:

```
extern int func();
static int func();
main()
{}
```

or:

```
extern int x = 3;  /* error requires initialization */
static int x;
```

Correct this error by deleting the incorrect declaration. This message may occur when other statements separate the `extern` and `static` declarations as long as the function or variable is not referenced.

`cc: 'tag' redeclared: saw struct or union tag, then enum tag`

The tags for `struct`, `union`, and `enum` type specifiers occupy the same name space; tag names for these type specifiers cannot be duplicated in the same scope.

Example:

```
enum bool {false, true};

main()
{
    struct bool { int value;};
    enum bool;
}
```

This code generates this message. The declaration of `bool` as a `struct` is legal because it is in a new block. This declaration hides the declaration of `bool` as an `enum` type in file scope. The second `enum` declaration of `bool` is illegal because it has already been declared to be a `struct` tag name in the block.

`cc: 'variable' redeclared: saw two initializers`

This message occurs when an identifier is declared and initialized more than once.

Example:

```
int i = 5;
int i = 4;
```

Correct this by deleting the incorrect initialization or declaration. There may be function definitions or other declarations between the two initializations.

`cc: 'identifier' redeclared: was previously declared as constant of enumerated type.`

An identifier declared as a constant of an enumerated type cannot be redeclared in the same scope as another type.

Example:

```
enum tf { true, false };
int true = 10;
main()
{}
```

Correct this error by renaming the `enum` constant or the variable.

Identifiers that are not label names, tags, or members of structures or unions occupy the same *name space.* Duplicate identifiers in the same name space are not allowed.

`cc: 'identifier' redeclared: was previously declared as typedef name.`

An identifier declared as a `typedef` name cannot be redeclared in the same scope as another type. An example that generates this error message is:

```
typedef int ident;

int ident = 5;
```

Correct this error by renaming either the `typedef` identifier or the variable.

Identifiers that are not label names, tags, or members of structures or unions occupy the same *name space.* Duplicate identifiers in the same name space are not allowed.

`cc: repeated value 'value' in select directive`

None of the parameters of the `select` pragma (directive) can have the same numerical value.

Example:

```
#pragma _CNX select(5,7,5)
```

means that both the vector and parallel-vector versions of a loop should be run when the trip count of a loop is five or six, is impossible. The parameters of the `select` pragma (directive) can never have the same value.

`cc: saw asm statement in 'file', optimization level reduced to -O0.`

This message indicates that the compiler encountered an **asm** statement in file *file*.

Example:

```
main()
{
    asm("dsi");
    asm("eni");
}
```

The compiler cannot optimize code that contains **asm** statements beyond optimization level -O0. Isolate functions that contain **asm** statements in separate source files, if possible, so that they do not prevent other functions from optimizing at a higher level.

`cc: variable set but not used in function function_name.`

Message Name: **set_but_not_used**

This message indicates that an **auto** variable declared in a function was assigned a value, but it was never used:

```
/* check with lint -d set_but_not_used=e file.c */
main()
{
    int i;

    i = 5;
}
```

This may indicate a logic error in the program.

`cc: shift by a negative value is undefined`

Message Name: **neg_shift**

This message indicates that the right operand of a shift operator is a negative constant.

Example:

```
/* compile with cc -d neg_shift=e file.c */
int x = 5;
x = x >> -1;
```

If the right operand of a left shift (<<) is negative, a right shift is performed; similarly, if the right operand of a right shift (>>) is negative, a left shift is performed. In such cases, the right shift or the left shift, uses the absolute value of the right operand.

```
cc: shift by a value greater than object size
```

Message Name: `shift_too_large`

This message indicates that the right operand of a shift operation exceeds the number of bit positions that the left operand can be displaced. The integral data types `char` and `short int` have the same bit count as the data types `int` and `long int` in this operation.

Example:

```
/* check with lint -d shift_too_large=e file.c */
main()
{
    char x = -5;
    x = x >> 32;
}
```

Correct this error by changing the data type to a larger size or reducing the right operand of the shift operator.

```
cc: signed long long bitfields are not supported
```

This message indicates that a `signed long long int` bit field has been defined.

Example:

```
/* compile with cc -d non_int_bit_field file.c */
struct {
    long long i:20;  /* ok - unsigned */
    signed long long j: 40;  /* not supported */
} some_s;
```

The `signed long long` type of bit field is not supported.

```
cc: sizeof(bitfield) disallowed
```

This message indicates a bit field is used as the operand of the `sizeof` operator. In ANSI C it is illegal to use a bit field as an argument of the `sizeof` operator.

`cc: sizeof(function) disallowed.`

The operand of the **sizeof** operator cannot be a function, but it may be a pointer to a function.

Example:

```
int func(int a);

int x = sizeof(func);
```

is illegal, while:

```
int (*func)(int a);

int x =  sizeof(func);
```

is not illegal.

`cc: sizeof(void) disallowed.`

void is not permitted as an operand of the **sizeof** operator.

Example

```
int k = sizeof(void);
```

Correct this error by replacing **void** with the correct data type.

`cc: static variable` *name* `unused`

This message indicates that variable *name* has been declared in file scope with the **static** storage class, but it is not used.

Example:

```
static int j;
main()
{
    static int no_err;
}
```

Correct this error by deleting the unused **static** variable.

`cc: storage class ignored - no declarators were declared`

Message Name: **class_ignored**

It is not useful to use an explicit storage class in the declaration of a structure. Although it is not illegal, the scope of the declaration does not extend to any variables declared with that **struct** type.

Example:

```
/* compile with cc -d class_ignored=e file.c */
static struct some_s {
    int x;
    int y;
};
struct some_s bad;
```

In this example, `bad` does not have `static` storage; the `static` storage class has no impact on `struct` declarations defined with `some_s`.

`cc: storage class specifier not allowed in this context`

This message indicates that a storage-class keyword appears in a context in which it is not allowed. The five storage-class keywords are `auto`, `extern`, `register`, `static`, and `typedef`. For example, storage-class specifiers cannot be used in the declaration of `struct` members:

```
struct x {
    typedef int number;
    register float x;
};
```

Correct this error by removing the `typedef` declarations, if any, or the storage-class specifiers.

`cc: string or character constant not terminated`

This message indicates that a string or character constant was not terminated by a closing double quote (") or a closing single quote ('), respectively.

Example:

```
#line 4 "s
#define x '23
```

Correct this error by including the closing double or single quote.

```
cc: struct name has no member information
```

Message Name: `incomplete_record`

This message indicates that the *name* `struct` was declared as a `struct`, but its members were not declared.

Example:

```
/* check with lint -d incomplete_record=w file.c */
struct empty_s;
```

Correct this situation by declaring the members of the `struct` or by deleting the declaration. This diagnostic option also detects empty `union` declarations.

```
cc: struct/union or struct/union pointer required.
```

This message occurs when the `.` operator is used instead of the `->` operator.

Example:

```
struct { int x; } *p;
int i;

i = p.x;
```

Because `p` is a pointer to a `struct`, use the `->` operator instead of the `.` operator in this example.

```
cc: A structure or union member may not have a function type.
```

The member of a structure or union can not be a function, but it can be a pointer to a function.

```
struct sample_s {
    int func( void );
};
```

generates the message, but

```
struct sample_s {
    int (*func)( void );
};
```

does not because it uses a pointer to a function instead of a function type.

```
cc: A structure or union member may not have type void.
```

A structure or union member cannot have type `void` because the compiler cannot determine how much memory to allocate for the member. The following example generates this message:

```
struct some_s {
    void x;
};
```

This error can be corrected by changing the `void` type to some other type, possibly `void *`.

```
cc: subscript not constant or constant expression.
```

This message occurs when a nonconstant expression specifies the size of an array. The constant expression must evaluate to an integer at compile time. Integral constant expressions include:

- Integer constants: decimal, octal, and hexadecimal
- Enumeration constants
- Character constants
- `sizeof` expressions
- Floating-point constants that are immediate operands of integral casts except as part of an operand to the `sizeof` operator

cc: subscript of array *array name* is not of an integer type

The expression contained within the `[]` operator must evaluate to an integer type. Thus, while

```
int b[10];
float c = 1.5;

b[ c ] = 5;
```

generates this message,

```
int b[10];
float c = 1.5;

b[ (int)c ] = 5;
```

does not generate any message.

```
cc: a switch case label must be integer.
```

A **case** label of a **switch** statement must be an integral type.

Example:

```
int expn = 3;

switch( expn ){
    case 3.0: ;
}
```

```
cc: a switch control expression must be integer.
```

The control expression of a **switch** statement must have an integral type.

Example:

```
main()
{
    float expn = 3.0;

    switch( expn )
        ;
}
```

```
cc: synch_parallel directive ignored - directive not allowed on
    outer loop
```

This message occurs when you did not use the **synch_parallel** optimization pragma on the innermost loop of code that has nested loops. The following code demonstrates incorrect use of this pragma:

```
main()
{
    int h,i,j;
    float a[20][300][10],b[20][300][10];

    for( h=0; h<20; h++ )
        #pragma _CNX synch_parallel
        for(j=0; j<10; j++)
            for(i=0; i<100; i++ )
                a[h][i+1][j] = a[h][i][j] + b[h][i][j];
}
```

Correct this error by applying the **synch_parallel** pragma to innermost loops only.

cc: Syntax Error at "*string*"

There is a syntax error in the indicated file at *string*. If other errors precede this error, the error may go away when you fix the preceding errors.

cc: Syntax Error at typedef name *identifier*

This is similar to an unqualified syntax error, but indicates the compiler has interpreted the *identifier* as a `typedef` name.

cc: syntax error in *directive* directive

There is a syntax error in the indicated pragma (directive). Refer to Appendix B, "Pragmas," for assistance with *directive*.

Syntax error recovery skipped to '*char*' *file*

Message Name: `skip_to_char`

This message indicates that the compiler is skipping to *char* in an attempt at error recovery. It will follow another syntax error message and does not require additional action to fix.

Syntax error recovery skipped to end of file

Message Name: `skip_to_eof`

This message indicates that the compiler is skipping to the next input file because it cannot recover from compilation errors. It will follow another syntax error message and does not require additional action to fix.

cc: tag redeclared: "union tag;" can't match any existing tag.

This message occurs when *tag* is identical to another tag (`enum`, `struct`, or `union`) declared previously in the same block scope.

Example:

```
int f(){
    union some_tag { true, false };
    union some_tag;
}
```

To avoid this error, rename *tag* with a tag name not used elsewhere in the block.

cc: tag redeclared: "struct *tag;*" can't match any existing tag.

This message occurs when *tag* is identical to another tag (enum, struct, or union) declared previously in the same block scope.

Example:

```
int f(){
    struct some_tag { true, false };
    struct some_tag;
}
```

To avoid this error, rename the *tag* with a tag name not used elsewhere in the block.

cc: tag '*tag_name*' redeclared.

There are three types of tags: struct, union, and enum. This message occurs when more than one declaration of a single tag is visible at one time in a source file.

Example:

```
enum color { red, white };
struct color {
    int hue;
    int intensity;
};
```

This code fragment generates this message. Correct the error by renaming one of the tags.

cc: '*option*' target machine not supported

This message occurs when an argument of the -tm command line option is not recognized.

Example:

```
cc -tm e1 file.c
```

This code generates this message because e1 is not a valid argument of the -tm command line option.

cc: There is no control path leading to a return from this function.

This message occurs when the compiler is unable to determine whether the indicated function has an exit.

Example:

```
int main()
{
start:
    exit(0);
    goto start;
}
```

This message does not prevent the compilation of a source file; it is a warning that may indicate a problem in the logic of a program.

`cc: too few arguments in call to function prototype`

This message indicates that the number of arguments in a function call does not agree with the number of arguments declared in its function prototype.

Example:

```
void func(int a, int b);

main()
{
    func(3);
}
```

If the discrepancy is intentional, use the macros provided in the stdargs.h include file.

`cc: too many arguments in call to function prototype`

This message indicates that the number of arguments in a function call does not agree with the number of arguments declared in its function prototype.

Example:

```
void func(int a, int b);

main()
{
    func(3, 4, 5);
}
```

If the discrepancy is intentional, use the macros provided in the stdargs.h include file.

`Too many errors to continue`

There are too many errors to permit the compiler to complete its examination of the source code. When the compiler reports an error, it makes an assumption about what the programmer intended before it continues compiling. If that assumption is incorrect, the compiler can report errors that don't exist. This *cascade effect* can prevent the compiler from checking the entire source file. The removal of the initial error can resolve subsequent errors.

`cc: too many initializers.`

This message occurs when an array, `struct`, or `union` is initialized with too many values.

Example:

```
char arry[5] = "123456";
```

In this declaration the array `arry` has five elements, but the string has 6 characters in it. *See also* `NULL at end of string initializer` *'characters'* `truncated.`

`cc: Translation unit contains no external declarations`

Message Name: `no_external_declaration`

Every translation unit (source file, compilation unit) must have at least one external declaration. This is an ANSI C requirement, but is not a requirement of CONVEX C.

`cc: the type in a function definition can not be provided by a typedef`

It is legal to declare a function type using `typedef`.

Example:

```
typedef int func_t(int param);
```

But such a function type cannot be used to *define* a function.

Example:

```
typedef int func_t(int param);

func_t func
{
}
```

is illegal. Resolve this message by defining the function with a type synonomous with the `typedef` name. Use the following example to resolve the problem with your code.

```
int func(int param)
{
}
```

cc: `type of function` *function name* `is not a function type`

This message occurs when the body of a function is improperly defined.

Example:

```
int func
{
}
```

In this example, the parameter list, including the parentheses, is missing. A set of parentheses must always precede the braces.

cc: `type pointed to by formal lacks some qualifiers of actual.`

Message Name: `arg_ptr_qual`

Qualifiers of a formal function parameter that is a pointer must be the same as those for an actual function parameter that is a pointer.

Example:

```
/* compile with cc -d arg_ptr_qual=e file.c */
extern int func1( int *x );

main()
{
    int a;
    const int  *b;

    a = func1( b );
}
```

The actual parameter has a `const` qualifier, while the formal parameter does not. This situation can be corrected by either declaring a `const` formal parameter or removing the `const` declaration on the actual parameter. *See also* **`type pointed to by formal lacks some qualifiers of actual`**.

`cc: Type pointed to by left operand of assignment must contain all qualifiers of type pointed to by right operand`

Pointers to `const`- or `volatile`- qualified data types may be assigned to pointers of the same qualified types. This message is caused by:

```
int const * x;
int *y;

y = x;
```

because **y** is not a `const`-qualified type. The following is legal:

```
int * x;
int const * y;

y = x;
```

Note that this requirement does not exist for `const`- or `volatile`-qualified pointers. The following is legal:

```
int * const x;
int * y;

y = x;
```

`cc: a typedef name may not have an initial value.`

Because a `typedef` name identifies a type, not a variable, it cannot be initialized. This message can be generated by:

```
typedef int var = 1;
```

To resolve this error, you must initialize the variable when it is declared or in the code itself.

Example:

```
typedef int var;
var x = 1;
```

`cc: unacceptable operand of &.`

Illegal operands of the & operator include:

- Bit fields in a structure
- Variables that have the `register` storage class
- Anything that is not an lvalue

These are objects that have no address, which is the function of the & operator.

`cc: unary & for array or function ignored.`

This message occurs only in the backward-compatible mode of the compiler. It results when the unary & operator is needlessly used. There are several ways to specify the base address of an array. For example, given the declaration

```
int arrx[10];
```

the base address of the array can be specified as one of the following in ANSI C:

- arrx /* type is int * */
- &arrx /* type is ptr to array of 10 ints */
- &arrx[0] /* type is ptr to int */

This message occurs when the second form, `&arrx`, is used. Similarly, there are several ways to specify the address of a function. For example, given the function prototype

```
int func( void );
```

the address of the function can be specified as one of the following:

- func
- &func

This message occurs when the second form is used, `&func`; the unary & operator is redundant in these examples.

`cc: unexpected end of file`

This message indicates that the compiler encountered the end of a file while it was processing a comment.

Example:

```
int j;
/* this comment doesn't end ...
```

Correct this error by terminating the comment causing the error.

`cc: Unexpected end of source file.`

The compiler encountered the end of the source file unexpectedly. Several causes of this error are listed below

- End of source file in a comment
- Mismatched pairs of { and }
- Mismatched pairs of (and)

The last two problems can be caused by misplaced comments or conditionally compiled code.

`cc: union` *name* `has no member information`

Message Name: `incomplete_record`

This message indicates that the *name* `struct` was declared as a `struct`, but its members were not declared.

Example:

```
/* check with lint -d incomplete_record=w file.c */
union empty_u;
```

Correct this situation by declaring the members of the `union` or by deleting the declaration. This diagnostic option also detects empty `struct` declarations.

`cc: unknown size for variable` *'variable name'*`.`

This message results when a declaration for a variable does not specify a storage size. For example,

```
int arrx[];
```

does not indicate how many elements are in array `arrx`. This is sometimes referred to as an incomplete type.

This message also occurs when a member of a `struct` definition refers to the tag of the `struct` that the member is declared in.

Example:

```
struct some_s {
    struct some_s bad;
};
```

This happens because the compiler does not know how much memory to allocate for `some_s`; its definition is not complete. Fix this situation by referring to the structure with a pointer.

Example:

```
struct some_s {
    struct some_s *correct;
};
```

While the structure `some_s` may be undefined at the time `correct` is declared, pointers have a known size.

`cc: Unrecognized diagnostic name 'identifier' -- ignored`

The compiler did not recognize the *identifier* specified with the `-d` command line option. The recognized diagnostic names are defined at the beginning of this appendix.

`cc: Unrecognized Escape Sequence 'sequence'`

Three types of escape sequences are recognized by the compiler:

- Character escape sequences including:
  \', \", \?, \, \a, \b, \f, \n, \r, \t, \v
- Octal escape sequences: \ followed by one, two, or three octal digits
- Hexadecimal escape sequences: **\x** followed by hexadecimal digits

The compiler interprets `\c` as `c` when `c` is not one of the recognized characters in a character escape sequence. This is an implementation-defined feature of the CONVEX C V4.0 compiler and may be different on another compiler. Refer to Chapter 3, "Compatibility Modes," for more information on escape sequences.

cc: unrecognized preprocessing directive *identifier*, skip to end of line

This message indicates that an unknown preprocessor directive was encountered.

Example:

```
# ifdefined x
```

Correct this error by deleting the unknown directive.

unrecognized preprocessing flag '*flag*', ignored

Message Name: **pp_badflag**

This message is only generated when the C preprocessor is run by itself with the **cpp** command. It indicates that you used an option not recognized by the preprocessor. You may have used a command line option recognized by the C compiler, but not recognized by the C preprocessor.

cc: *name* unused in function *func*

This message indicates that function *func* declares *name* as a local variable, but does not use it in its function definition.

Example:

```
int func( int arg1 )
{
    int y;
    return (arg1);
}
```

Correct this error by removing the unused local variable from the function definition.

cc: '*identifier*' used before being assigned.

This message indicates that *identifier* is used in an expression before it has been assigned a value.

Example:

```
#include <stdio.h>
main()
{
    int j;
    int a[10];

    a[j] = 5;
    printf("a[%d] = %d\n", j, a[j] );
}
```

Only `static` variables are assigned a default value of 0. Correct this error by initializing *identifier* before you use it.

```
Using -lc in -pcc mode causes incorrect library usage
```

The `-lc` switch causes files to be linked with a library supplied for use with the extended mode of the compiler. Using it in the backward-compatible mode can cause linkage errors. In most cases removing `-lc` from the command line allows linking to proceed normally and produce the correct result; `cc` automatically searches the correct library sets, so using `-lc` is not necessary.

```
cc: VARARGS applied to prototype, use '...'.
```

Message Name: `varargs_on_proto`

This message occurs when the VARARGS directive precedes a function definition that has a function prototype declared for it.

Example:

```
/* check with lint -d varargs_on_proto=e file.c */
/*VARARGS*/
int f( int a, int b);

int f( int a, int b )
{
    return (1);
}
```

There are two ways to remove this error message. You can declare the function prototype using the '...' syntax.

```
int f( int a, ... );

int f( int a, int b )
{
    return (1);
}
```

Or, you can compile with the -d varargs_on_proto command line option.

cc: VARARGS greater than number of formals.

Message Name: varargs_too_large

This message occurs when the argument of the **VARARGS** lint directive is greater than the number of formal arguments in the function that follows it.

Example:

```
/* check with lint -d varargs_too_large=e file.c */
/*VARARGS3*/
int f( a, b )
int a,b;
{
    return (1);
}
```

Correct this error by changing the reducing the value of **VARARGS**'s argument to a number smaller than the number of formal arguments.

cc: variable '*variable name*' can not be declared 'auto' in file scope.

The auto storage class can be used only for local variables. The following program:

```
auto int z;
main()
{
}
```

generates this message.

Correct this error by removing the auto storage class from the declaration of the file scope variable.

cc: variable '*variable name*' can not be declared '*register*' in file scope.

The register storage class can be used only for local variables and declaration of formal parameters. The following code illustrates this type of error.

Example:

```
register int z;
main()
{
}
```

Correct this code by removing the `register` keyword because the compiler automatically optimizes register usage.

`cc: variable 'var_name' redeclared.`

This message occurs when an identifier is declared more than once within one block of code. A block of code is delimited by { and }. This includes the definition of a `struct` or a `union`.

Example:

```
void func()
{
    int x;
    float x;
}
```

This condition can be corrected by deleting the inappropriate declaration or renaming one of the variables.

`cc: variable 'var_name' undefined.`

This message occurs when an identifier is used in a context where its declaration is not visible. Either the identifier is not defined in the block in which it is used, or it is not defined with file scope prior to the block that it is used in.

Example:

```
void func()
{
    x = 5;
}
```

To resolve this, define the variable in a location that is visible to the block it is used in.

`cc: vector directive inside vector directive ignored`

This message indicates that a vector optimization directive or pragma that inside of a loop that you have directed the compiler to vectorize.

Example:

```
#include <stdio.h>
main()
{
    int i,j,sum=0;
    int a[100][100];

    #pragma _CNX force_vector
    for(i=0; i<100; i++)
        #pragma _CNX force_vector
        for(j=0; j<100; j++)
            sum += a[i][j];
    printf("sum = %d0, sum );
}
```

The `force_vector` pragma that precedes the second loop causes this error message to occur. To correct this error, remove any vectorization pragmas or directives that occur in loops preceded by another vectorization pragma or directive.

`cc: void type not allowed in this declaration.`

The `void` type cannot be used to declare variables.

Example:

```
void var;   /* or */
void a[5];  /* or */
int func(void param);
```

Typically, `void` is used as the return type of a function, the base type of a pointer, or to indicate that a function has no parameters.

`cc: void type used in expression.`

Declaring the elements of an array as type `void`, or defining function parameters as type `void` is illegal.

Example:

```
void a[5];  /* or */
int func(void param);
```

A declaration of type `void` is illegal in these cases because the compiler cannot determine the amount of memory to allocate for the identifiers.

`cc: wrong number of actual arguments to macro 'name'`

Message Name: `pp_argcount`

This message indicates that the number of arguments in the actual argument list does not match the number of arguments in the formal argument list of a function-like macro, except in the case where there is only one formal argument and no actual argument.

Example:

```
/* compile with cc -d pp_argcount=w file.c */
#define a(b,c,d) b+c+d
int x = a(1,2);
int y = a(1,2,3,4);
```

Correct this error by specifying the correct number of actual arguments.

`cc: zero field size for field 'member name'.`

Bit fields with a width of zero may only be used with unnamed structure members. Such members are used to pad the structure to the next word alignment. In the following code, the declaration of member **x** is illegal:

```
struct {
    int p:3;
    int x:0;
    int y:5;
} bad;
```

Correct this error by placing zero bit fields at the end of a structure definition.

`cc: zero length character constant.`

Character constants must contain at least one character in a character sequence. The following is illegal:

```
int ch = '';
```

```
cc: [] requires a pointer or array type.
```

This message occurs when none of the operands of the array subscript operator, `[]`, is a pointer or an array name. The following example generates this error message:

```
main()
{
     int i,j,a[20];

     i[j] = 5;
}
```

When you use the array subscript operator, `[]`, one of the operands must be a pointer or array name, and the other must be an integer. Correct this error by ensuring that one operand is a pointer or array name.

`cc: 'unsigned` *comparison-operator* `0' can be simplified`

Message Name: `uns_compare_zero`

This message indicates that a comparison operation between an unsigned number and zero is not necessary. The comparison operator cannot be == or !=.

Example:

```
/* check with lint -d uns_compare_zero=w file.c */
unsigned x;
if( x >= 0 ) x++;
```

In this case, the expression **x >= 0** can be replaced by the expression **x**. Similar replacements can be performed for the <, >, and <= operators.

cc: 'unsigned *comparison-operator* negative constant' is constant

Message Name: uns_compare_neg

This message indicates that a comparison is being made between a short unsigned integer and a negative constant.

Example:

```
/* check with lint -d uns_compare_neg=w file.c */
c()
{
    short unsigned x;
    if( x <= -1 );
}
```

Because **x** is unsigned, it is always greater than any negative constants. Consequently, the expression can be replaced by a constant Boolean value, such as **1**.

Index

& (address-of operator) 8-73, 8-75, 8-76, A-145, G-238, G-274, G-313
() (function) operator, operand of F-202, G-285
–, in %[scanlist, no special meaning E-176, E-184
/**/ (pasting) operator 3-36
??! trigraph of | 3-35
??' trigraph of ^ 3-35
??(trigraph of [3-35
??) trigraph of] 3-35
??- trigraph of \(ap 3-35
??< trigraph of { 3-35
??= trigraph of # 3-35
??> trigraph of } 3-35
??/ trigraph of \ 3-35
– (unary minus) operator F-253, G-285
+ (unary plus) operator F-253, G-285

A

abort function
  action on open files E-180, E-189
  action on temporary files E-180, E-189
  description 11-119
abs function 11-119
acos function
  description 11-102
  returns acos(1) on domain error E-173, E-181
Ada routines, calling 8-78
adb debugger 5-53
aggregate
  definition F-206, G-233
  initialization of automatic storage class F-206, G-233
alignment of data types E-171, E-179
ANSI C
  changes preventing compilation 3-34
  compatibility mode features 3-29
  future directions 3-37
  header file changes 3-36
  semantic changes 3-35
apparent recurrence B-153
append mode
  location of file position indicator E-176, E-184
argc, argument of function main E-167, E-175
arg_ptr_qual
  compatibility mode actions F-196, G-228
  description F-192, G-224
  error message example F-276, G-311
arg_ptr_ref
  compatibility mode actions F-196, G-228
  description F-192, G-224
  error message example F-202, G-230
argument pointer 8-69
argv, argument of function main E-167, E-175
arithmetic
  conversions 3-35
  expressions with float 3-36
array
  addresses A-145, A-153
  alignment E-171, E-179
  character array initialization F-251, F-275, G-282, G-310
  data type A-145, A-153
  data type of index E-170, E-178
  dimensions A-145, A-153
  zero-length 3-34
asctime function 11-126
asin function
  description 11-102
  returns asin(1) on domain error E-173, E-181
-asm, compiler option 13-133
asm keyword 13-133
asm statement 3-34, 13-133
assembly-language debugger 5-53
assembly-language statements 13-133
_assert function 11-93
assert function
  description 11-93
assert macro
  form of message displayed E-172, E-180
  terminated by abort function E-172, E-180
assert.h contents 11-93
assign_in_condition
  compatibility mode actions F-196, G-228
  description F-192, G-224
  error message example F-205, G-233
associated documents, xiv
  how to order xv

atan function
description 11-102
returns $\pi/2$ on domain error E-173, E-181
atan2 function
description 11-102
returns $\pi/2$ on domain error E-173, E-181
atexit function 11-119
atof function 11-104, 11-117
atoi function 11-117
atol function 11-118
auto storage class, legal uses F-282, G-318

B

backup C compiler 3-30
backward-compatible mode
converting to 3-38
example 3-29
long long int A-138, A-146
overview 3-27
porting to 3-30
basic block
optimization 7-61
begin_tasks pragma B-151
bibliography xiv
bint.h.3bit man page F-214
bit field
alignment E-171, E-179
allocation E-171, E-179
data types F-236, G-266
initialization F-239, G-269
maximum size F-225, G-254
order of allocation E-171, E-179
sign of E-171, E-179
straddling byte boundaries E-171, E-179
zero width F-285, G-321
bitwise operation
on a signed integer E-169, E-177
on an unsigned integer E-169, E-177
blocking of signals E-175, E-183
bsearch function 11-119
BUFSIZ macro 11-110

C

cabs function 11-104
calling sequence, standard 8-70
calling
Ada routines 8-78
FORTRAN functions 8-77
runtime functions 11-93
calloc function
description 11-118
request size zero E-180, E-188
calls, standard function 8-73
case, maximum number E-172, E-179
casting, pointers and integers E-170, E-178
caution
force_parallel_ext pragma B-151, B-152, B-159, B-160
force_vector pragma B-152, B-160
no_recurrence pragma B-153, B-161
cc
command line 3-30
man page F-182
CCLIBS environment variable 2-22
CCOPTIONS environment variable 2-22
ceil function 11-102
char data type
alignment E-171, E-179
description A-135, A-143, A-147, A-155
range of values E-169, E-177
character array
initialization F-251, F-275, G-282, G-310
character constant
initialization F-211, G-239
maximum length F-211, G-239
sign in preprocessor conditionals E-172, E-180
value E-168, E-176
wide F-241, G-272
zero length F-285, G-321
character data A-146, A-147, A-154, A-155
character input and output functions 11-113
character representation
eight bits E-168, E-176
character set
execution E-168, E-176
illegal characters F-233, G-263
source E-168, E-176
CHAR_BIT macro 11-98
char_cvt_truncates
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-214, G-242
CHAR_MAX macro 11-98
CHAR_MIN macro 11-98
CHILD_MAX macro 11-100
class_ignored
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-267, G-302
clearerr function 11-115
CLK_TCK macro 11-126

clock function 11-125
CLOCKS_PER_SEC macro 11-124
clock_t type 11-124
Common C compiler
  converting from 3-30
  incompatibilities 3-38
  permits illegal code 3-38
communication with the environment 11-119
comparison functions 11-122
compatibility modes
  backward-compatible 3-27
  conforming 3-27
  default 3-27
  description 3-27
  differences 3-27
  examples 3-28
  extended 3-27
  features 3-29
  strict 3-27
compatibility, object file 2-25
compatible type
  description F-254, G-286
  pointer F-254, G-286
compiler directives C-161, C-169
  See also pragmas
compiler error
  example 2-23
  internal compiler error F-212, G-240
  machine serial number mismatch F-211, G-238
  no available memory F-249, G-280
  parser stack exceeded F-198, G-293
  program is too complex F-261, G-293
compiler limits
  CPU time limit F-216, G-244
  file size limit F-226, G-255
  maximum number of scalars in function F-246, G-276
compiler pragmas B-149
  See also pragmas
compiler
  backup C 3-30
  Common C 3-30
  Common C, converting from 3-30
  definition 1-1
  mixed compatibility modes 2-24
compiling examples
  backward-compatible mode 3-29
  conforming mode 3-29
  default mode 3-28
  extended mode 3-28
  mixed mode 3-30
  strict mode 3-29
compiling
  conditional 12-131
  introduction 1-1
  language specifications 3-29
  library systems 3-30
  multiple files 1-3
  multiple modes 3-29
  one source file, example 1-2
  simple program 1-1
  single mode 3-28
concatenation functions 11-121
conditional compilation 12-131
conforming compatibility mode
  description 3-27
  example 3-29
constant
  \a 3-35
  \x 3-35
constants
  integer 3-35
  octal 3-35
const_condition
  compatibility mode actions F-196, G-228
  description F-192, G-224
  error message example F-213, G-241
const_not_init
  compatibility mode actions F-196, G-228
  description F-192, G-224
  error message example F-212, G-240
contact utility, documentation xiv
conversions
  float to double E-170, E-177
  float truncation E-170, E-178
  int to float E-170, E-177
  integral E-169, E-177
converting
  function pointers 3-34
  object pointers 3-34
CONVEX Consultant debugger 5-50
CONVEX extensions
  assert.h 11-93
  ctype.h 11-95
  math.h 11-104
  signal.h 11-107
  stdio.h 11-116
  stdlib.h 11-124
__convex__ macro 2-20
convex macro 2-20
CONVEX symbolic debugger 5-50
__convexc__ macro 2-21
_CONVEX_FLOAT_ macro 2-20
ConvexOS considerations 2-22
_CONVEX_SOURCE macro 2-21

use 3-28, 3-29
convexvc macro 2-21
copying functions 11-121
cos function
description 11-103
returns cos(0) on domain error E-173, E-181
cosh function
description 11-103
returns HUGE_VAL on domain error E-173, E-181
_count
function description D-164, D-172
man page F-217
cpp man page F-198
cref program 5-52
cross-reference generator 5-52
csd debugger 5-50
ctime function 11-126
ctype.h
ANSI C changes 3-37
contents 11-94
future changes 3-38
cuserid function 11-116
customer support
telephone number for xv
cvt_changes_sign
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-214, G-242
cvt_to_unsigned
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-216, G-244
CXdb Debugger 5-49
CXpa profiler 6-57

D

data representations
FORTRAN 8-74
data type
array A-145, A-153
char A-135, A-143, A-147, A-155
double A-139, A-147
enum A-138, A-146
float A-139, A-147
int A-135, A-143
long A-135, A-143
long double A-139, A-147
long float A-141, A-149
long int A-135, A-143
long long A-137, A-145
long long int A-137, A-145
pointer A-142, A-150
short A-135, A-143
short int A-135, A-143
string A-146, A-154
void A-143, A-151
__DATE__ macro 2-21, E-172, E-180
DBL_DIG macro 11-96
DBL_EPSILON macro 11-96
DBL_MANT_DIG macro 11-96
DBL_MAX macro 3-37, 11-96
DBL_MAX_10_EXP macro 11-96
DBL_MAX_EXP macro 11-96
DBL_MIN macro 11-96
DBL_MIN_10_EXP macro 11-96
DBL_MIN_EXP macro 11-96
dcvtid function 11-104
debugger
adb 5-53
assembly-language 5-53
consultant package 5-50
CONVEX CXdb 5-49
csd 5-50
symbolic 5-50
tools 5-49
declarators
minimum number of E-171, E-179
default compatibility mode
description 3-27
example 3-28
#define 12-127
diagnostic message, controlled
default settings F-196, G-228
diagnostic messages 2-23
difftime function 11-125
direct input and output functions 11-114
directives
compiler C-161, C-169
lint utility 4-44
See also pragmas
disabling intrinsic functions 9-81
div function 11-120
division_by_zero
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-212, G-239
division, integer, sign or remainder E-169, E-177
div_t type 11-117
documentation
ordering xv
subscription service, how to apply xv

dollar_names
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-221, G-249
domain error, math function return values E-173, E-181
double
alignment E-171, E-179
data type A-139, A-140, A-147, A-148
double-precision floating point A-140, A-148
_dshiftl
function desciption D-172, D-164
man page F-221
_dshiftr
function description D-164, D-172
man page F-221
dump, postmortem 5-50

E

EDOM
macro 11-95
use 11-101
empty declarations
struct 3-34, 3-36
union 3-34
end_tasks pragma B-151
–Enoposix, linker option 2-22
entry statement 3-34
enum
data type A-138, A-146
representation of E-171, E-179
environment, ConvexOS 2-22
environment variables
CCLIBS 2-22
CCOPTIONS 2-22
envp, argument of function main E-167, E-175
EOF macro 11-110
–ep option B-159
–Eposix, linker option 2-22
ERANGE
macro 11-95
use 11-101
errno
use 11-104
values of fgetpos and ftell E-176, E-185
variable 11-96
errno.h
contents 11-95
future changes 3-38
error handling functions 11-115
error message
catalog F-197, G-229
control F-191, G-223
–d options F-192, G-224
diagnostic options F-191, G-223
error message, controlled
arg_ptr_qual F-196, G-228
arg_ptr_ref F-196, G-228
assign_in_condition F-196, F-205, G-228, G-233
char_cvt_truncates F-196, F-214, G-228, G-242
class_ignored F-196, G-228
const_condition F-196, F-213, G-228, G-241
const_not_init F-196, G-228
cvt_changes_sign F-196, F-214, G-228, G-242
cvt_to_unsigned F-196, F-216, G-228, G-244
division_by_zero F-196, G-228
dollar_names F-196
escape_range_sequence F-196, G-228
eval_order F-196, F-256, G-228, G-288
float_suffix F-196, G-228
function_parameter F-196, F-227, G-228, G-257
function_pointer_cast F-196, G-228
hidden_arg F-196, F-204, G-228, G-231
hidden_extern F-196, F-224, G-228, G-252
hides_outer F-196, F-231, G-228, G-261
implicit_decl F-196, G-228
incomplete_record F-196, F-268, F-278, G-228, G-304, G-314
inefficient_alignment_efficiency F-196
int_cvt_truncates F-196, F-215, G-228, G-243
integer_overflow F-196, G-228
long_long_suffix F-196, G-228
misplaced_lint_directive F-196, G-228
negative_to_uns F-196, F-248, G-228, G-279
neg_shift F-196, G-228
no_arg_type F-197, F-228, G-228, G-258
no_external_declaration F-197, G-228
non_int_bit_field F-197, G-228
nothing_declared F-197, G-228
null_effect_expression F-197, G-228

pointer_alignment_efficiency F-197, G-228
pp_argcount F-197, F-285, G-228, G-321
pp_argsended F-197, G-228
pp_badflag F-232, F-280, G-262, G-316
pp_badkfile F-247, G-279
pp_badstr F-197, F-240, G-228, G-270
pp_badtp F-197, F-240, G-228, G-270
pp_extra F-197, F-225, G-228, G-254
pp_idexpected F-197, G-228
pp_line_range F-197, G-228
pp_macro_arg F-197, F-245, G-228, G-275
pp_macro_redefinition F-197, F-245, G-229, G-276
pp_macro_redefinition_cmdl G-229
pp_malformed_directive F-197, G-229
pp_old_dir F-197, F-252, G-229, G-284
pp_parse F-197, G-229
pp_undef F-197, G-229
pp_undef_cmdl F-197, G-229
pp_unrecognized_directive F-197, G-229
pp_unrecognized_pragma F-197, F-260, G-229, G-292
set_but_not_used F-197, G-229
shift_too_large F-197, G-229
short_cvt_truncates F-197, F-215, G-229, G-243
skip_to_char F-197, F-272, G-229, G-307
skip_to_eof F-197, F-272, G-229, G-307
strict_syntax F-197, G-229
uns_compare_neg F-197, F-286, G-229, G-323
uns_compare_zero F-197, F-286, G-229, G-322
unsigned_suffix F-197, G-229
varargs_on_proto F-197, G-229
varargs_too_large F-197, G-229
void_pointer_cast F-197, G-229
error utility 4-47
escape sequences
description F-279, G-315
value of undefined E-168, E-176
escape_range_sequence
compatibility mode actions F-196, G-228
description F-192, G-224
error message example F-223, G-252
eval_order
compatibility mode actions F-196, G-228
description F-193, G-225
error message example F-256, G-288
example
array of pointers to functions F-205, G-232
const-qualifier F-276, G-312
function parameter is pointer to function F-227, G-257
function returns pointer to array of int F-230, G-259
function returns pointer to functions returning int F-230, G-260
incomplete type F-204, G-232
tasking pragmas (directives) F-206, G-234
exit function
description 11-119
return value E-180
EXIT_FAILURE macro 11-117
EXIT_SUCCESS macro 11-117
exp function 11-103
expression evaluation order 3-35
extended compatibility mode
description 3-27
differences with Common C 3-33
example 3-28
porting to 3-31
extern storage class
changing from extern to static F-201, G-297
changing to static from extern F-198, F-263, G-294, G-297
initialization in function F-200, G-247

F

fabs function 11-103
fclose function 11-112
fdopen function 11-116
feof function 11-115
ferror function 11-115
fflush function 11-112
fgetc function 11-113
fgetpos function 11-115
fgets function 11-113
figure
char representation A-136, A-144
character data representation A-147, A-155
character string representation A-147, A-155
compiler and linker interactions 1-5
double-precision floating representation A-141, A-149
int representation A-137, A-145
linking library routines 1-6
long long representation A-138, A-146
multiple source files 1-4

pointer representation A-142, A-150
role of the compiler 1-1
sample program, file2.c 1-3
sample program, prog2.c 1-3
short int representation A-136, A-144
single-precision floating representation A-140, A-148
stack layout 8-72
top of the runtime stack 8-70
file access functions 11-112
file input and output
concepts 10-83
support for fully buffered E-176, E-184
support for line-buffered E-176, E-184
support for unbuffered E-176, E-184
file location
meaning of #include delimiters E-172, E-180
__FILE__ macro 2-21, 3-34
file manipulation 10-83
file name
length of E-176, E-184
naming conventions 2-8
file positioning functions 11-115
file types and access modes 10-85
FILE
description of 10-84
type 11-111
FILENAME_MAX macro 11-110
fileno function 11-116
flags
code generation 2-12
compatibility 2-9
diagnostic 2-15
miscellaneous 2-20
optimization 2-17
preprocessor 2-11
utility support 2-15
See also options
float
alignment E-171, E-179
data type A-139, A-147
promotion to double 3-36
float.h contents 11-96
floating-point range
double A-139, A-147
float A-139, A-147
long double A-139, A-147
long float A-142, A-150
floating-point
32-bit A-139, A-147
64-bit A-139, A-147
data types A-139, A-147
IEEE format A-139, A-147
native format A-139, A-147
product reduction operator B-156
sum reduction operator B-156
float_suffix
compatibility mode actions F-196, G-228
description F-193, G-225
error message example F-226, G-256
floor function 11-103
FLT_DIG macro 11-97
FLT_EPSILON macro 11-97
FLT_MANT_DIG macro 11-97
FLT_MAX macro 11-97
FLT_MAX_10_EXP macro 11-97
FLT_MAX_EXP macro 11-97
FLT_MIN macro 11-97
FLT_MIN_10_EXP macro 11-97
FLT_MIN_EXP macro 11-97
FLT_RADIX macro 11-97
FLT_ROUNDS macro 11-97
fmod function
description 11-103
domain error E-173, E-181
fopen function 11-112
FOPEN_MAX macro 11-110
force_parallel pragma B-151
force_parallel_ext pragma B-151, B-152
force_vector pragma B-152
formal parameters, typedef names 3-34
formatted input and output functions 11-113
FORTRAN, mixing I/O with C 8-78
fpos_t type 11-111
fprintf function
description 11-113
example 10-84
meaning of %p E-176, E-184
fputc function 11-114
fputs function 11-114
frame pointer 8-69
fread function 11-114
free function 11-118
freopen function 11-112
frexp function 11-103
fscanf function
description 11-113
meaning of %p E-176, E-184
fseek function parameter 11-111
fseek function
associated macros 11-111
description 11-115
fsetpos function 11-115
ftell function 11-115
function

arguments 8-73
definition 1-2
incomplete return type F-228, G-258
initialization F-230, G-259
parameter storage class F-236, G-266
stack layout 8-69
static storage class F-229, G-259
storage class F-235, G-265
function calls, invariant B-154
function names 8-73
accessing function names from C 8-74
function pointers
converting 3-34
different types 3-34
function-level optimization 7-62
function-like macros versus functions 11-92
function_parameter
compatibility mode actions F-196, G-228
description F-193, G-225
error message example F-227, G-257
function_pointer_cast
compatibility mode actions F-196, G-228
description F-193, G-225
error message example F-258, G-290
lint error message example F-258, G-290
functions
character input and output 11-113
close 3-37
communication with the environment 11-119
compared to function-like macros 11-92
comparison 11-122
concatenation 11-121
copying 11-121
creat 3-37
direct input and output 11-114
error-handling 11-115
fdopen 3-37
file access 11-112
file positioning 11-115
fileno 3-37
formatted input and output 11-113
implicit declaration 11-93
integer arithmetic 11-119
lseek 3-37
main E-167, E-175
memory management 11-118
miscellaneous 11-123
multibyte character 11-120
multibyte string 11-120
open 3-37
operations on files 11-112
pseudo-random sequence generation 11-118
read 3-37
sacos 3-37
sasin 3-37
satan 3-37
satan2 3-37
scabs 3-37
scos 3-37
scosh 3-37
search 11-123
searching and sorting 11-119
sexp 3-37
sfabs 3-37
shypot 3-37
signal 3-37
signal handlers 3-37
SIGNALS_IN_PROG 3-37
slog 3-37
spow 3-37
ssin 3-37
ssinh 3-37
ssqrt 3-37
stan 3-37
stanh 3-37
string conversion 11-117
string handling 11-121
time 11-124
time conversion 11-126
time manipulation 11-125
unlink 3-37
variable number of parameters 3-34
write 3-37
further reference xiv
fwrite function 11-114

G

gamma function 11-104
_gbit
function description D-164, D-172
man page F-216
_gbits
function description D-164, D-172
man page F-216
generic pointer A-143, A-151
getc function 11-114
getchar function 11-114
getenv function
description 11-119
set of environment names E-180, E-189
gets function 11-114
gmtime function 11-126

H

header files
  assert.h 11-93
  ctype.h 3-37, 11-94
  errno.h 11-95
  float.h 11-96
  limits.h 11-98
  locale.h 11-100
  math.h 3-37, 11-101
  setjmp.h 11-105
  signal.h 3-37, 11-106
  stdarg.h 11-109
  stddef.h 11-110
  stdio.h 3-37, 11-110
  stdlib.h 11-117
  string.h 11-121
  strings.h 3-37
  time.h 11-124
  use of 11-93
hidden_arg
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-204, G-231
hidden_extern
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-224, G-252
hides_outer
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-231, G-261
HUGE macro 3-37
HUGEI macro 3-37
HUGE_VAL macro 11-102
hypot function 11-104

I

IBCLR man page F-220
IBITS man page F-220
IBSET man page F-220
idcvtd function 11-104
identifiers
  external E-168, E-176
  internal E-168, E-176
  significant characters E-168, E-176
IEEE format
  double-precision A-141, A-149
  single-precision A-140, A-148
_IEEE_FLOAT_ macro 2-21
implicit function declaration 11-93
implicit_decl
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-239, G-269
#include
  meaning of delimiters E-172, E-180
  statement 12-129
incompatibilities of common C 3-38
incomplete type
  definition F-204, G-232
  example F-204, F-279, G-232, G-314
incomplete_record
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-268, F-278, G-304, G-314
indent utility 4-46
index function 11-124
inefficient_alignment_efficiency
  compatibility mode actions F-196
inhibitors of parallelization 7-64
inhibitors of vectorization 7-64
initialization
  character array F-251, G-282
  character string F-275, G-310
  function F-230, G-259
  legal initializations F-233, G-263
  struct 3-36
  typedef variables F-277, G-312
initializers, bit field F-239, G-269
__INLINE_MATH 9-79
input and output
  mixing C and FORTRAN 8-78
  program 10-86
int
  alignment E-171, E-179
  data type A-135, A-143
int_cvt_truncates
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-215, G-243
integer arithmetic functions 11-119
integer constants 3-35
integer division, sign of remainder E-169, E-177
integer range
  char A-136, A-144
  enum A-138, A-146
  int A-136, A-144
  long A-136, A-144
  long int A-136, A-144
  long long A-137, A-145
  long long int A-137, A-145

short A-136, A-144
short int A-136, A-144
integer representation
16-bit A-135, A-143
32-bit A-135, A-143
8-bit A-135, A-143
two's complement E-169, E-177
integer type A-135, A-143
integer_overflow
compatibility mode actions F-196, G-228
description F-193, G-225
error message example F-212, G-240
integral constant expressions, defined F-270, G-305
integral conversions E-169, E-177
interactive devices, location E-167, E-175
interchange, loop B-156
INT_MAX macro 11-98
INT_MIN macro 11-98
intrinsic functions 9-79
advantages 9-79
disabling 9-81
disadvantages 9-79
generation of signals 9-80
optimization of errno 9-81
signal handler 9-82
intrinsic instructions 9-79
errno 9-80
intro.3bit man page F-212
invariant function calls B-154
_IOFBF macro 11-110
_IOLBF macro 11-111
_IONBF macro 11-111
ipc/network operational error messages E-179, E-188
ipow function 11-105
ircvtr function 11-105
isalnum function
description 11-94
LC_CTYPE 11-101
returns true for E-172, E-180
isalpha function
description 11-94
LC_CTYPE 11-101
returns true for E-172, E-180
isascii macro3-37
description 11-95
iscntrl function
description 11-94
LC_CTYPE 11-101
returns true for E-172, E-180
isdigit function
description 11-94
LC_CTYPE 11-101
isgraph function
description 11-94
LC_CTYPE 11-101
islower function
description 11-94
LC_CTYPE 11-101
returns true for E-172, E-180
isprint function
description 11-94
LC_CTYPE 11-101
returns true for E-172, E-180
ispunct function
description 11-94
LC_CTYPE 11-101
isspace function
description 11-94
LC_CTYPE 11-101
isupper function
description 11-94
LC_CTYPE 11-101
returns true for E-172, E-180
isxdigit function
description 11-94
LC_CTYPE 11-101

J

j0 function 11-105
j1 function 11-105
jmp_buf type 11-105
jn function 11-105

K

kill function 11-108

L

label, scope of F-221, G-250
labs function 11-120
LC_ALL macro 11-100
LC_COLLATE macro 11-100
LC_CTYPE macro 11-101
LC_MONETARY macro 11-101
LC_NUMERIC macro 11-101
lconv structure 11-100
L_ctermid macro 11-116
LC_TIME macro 11-101
L_cuserid macro 11-116
LDBL_DIG macro 11-97

LDBL_EPSILON macro 11-97
LDBL_MANT_DIG macro 11-97
LDBL_MAX macro 11-98
LDBL_MAX_10_EXP macro 11-98
LDBL_MAX_EXP macro 11-98
LDBL_MIN macro 11-98
LDBL_MIN_10_EXP macro 11-98
LDBL_MIN_EXP macro 11-98
ldexp function 11-103
ldiv function 11-120
ldiv_t type 11-117
_ldzero
  function description D-164, D-172
  man page F-217
_leadz
  function description D-164, D-172
  man page F-217
libraries, general 1-5
limits.h contents 11-98
__LINE__ macro 2-21, 3-34
linker
  description 1-4
  specifying libraries 2-22
  usage 2-22
linker options
  -Enoposix 2-22
  -Eposix 2-22
  on cc command line 2-22
LINK_MAX macro 11-100
lint error message
  function_pointer_cast F-258, G-290
  misplaced_lint_directive F-247, G-278
  neg_shift F-265, G-300
  pointer_alignment_efficiency F-257, G-289
  set_but_not_used F-265, G-300
  shift_too_large F-266, G-301
  varargs_on_proto F-281, G-317
  varargs_too_large F-282, G-318
  void_pointer_cast F-259, G-291
  See also error message
  See also error messages, controlled
lint utility
  converting to backward-compatible mode 3-31, 3-32
  description 4-43
  directives 4-44
  man page F-203
  options 4-44
locale
  available E-169, E-177
localeconv function
  description 11-101
  LC_MONETARY 11-101
  LC_NUMERIC 11-101
locale.h
  contents 11-100
  future changes 3-38
localtime function 11-126
log function
  description 11-103
  returns log(|x|) on domain error E-173, E-181
log10 function
  description 11-103
  returns log10(|x|) on domain error E-173, E-181
long
  alignment E-171, E-179
  bit shift operand 3-36
  data type A-135, A-143
long double
  alignment E-171, E-179
  data type A-139, A-147
long float, data type 3-34, A-141, A-149
long int data type A-135, A-143
long long data type A-137, A-145
long long int
  data type A-137, A-145
  function parameter, backward-compatible mode A-138, A-146
long_long_suffix
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-241, G-271
longjmp function
  description 11-106
  optimization with F-255, G-287
LONG_MAX macro 11-98, 11-99
LONG_MIN macro 11-98
loop interchange B-156
loop-carried dependency 7-65
lpow function 11-105
L_tmpnam macro 11-111
lvalue, definition of F-243, G-273

M

machine-dependent optimizations 7-60
macro
  parameter substitution 3-36
  recursive 3-36
main function E-167, E-175
make utility 4-45

malloc function
  description 11-118
  request size zero E-180, E-188
man pages
  bint.h include file F-214
  bint_t typedef F-214
  cc F-182
  _count function F-217
  cpp F-198
  _dshiftl function F-221
  _dshiftr function F-221
  _gbit function F-216
  _gbits function F-216
  IBCLR macro F-220
  IBITS macro F-220
  IBSET macro F-220
  intro.3bit F-212
  _ldzero function F-217
  _leadz function F-217
  lint F-203
  _mask function F-219
  _maskl function F-219
  _maskr function F-219
  MVBITS macro F-220
  __NO_INLINE_BINT F-214
  _parity function F-217
  _pbit function F-216
  _pbits function F-216
  _popcnt function F-217
_mask
  function description D-164, D-172
  man page F-219
_maskl
  function description D-164, D-172
  man page F-219
_maskr
  function description D-164, D-172
  man page F-219
math domain error return values 11-104
math functions
  setting of errno E-173, E-181
  underflow E-173, E-181
  values returned on domain errors E-173, E-181
math.h
  ANSI C changes 3-37
  contents 11-101
  future changes 3-38
MAX_CANON macro 11-100
MAX_INPUT macro 11-100
max_trips pragma B-153
MB_CUR_MAX macro 11-117
mblen function 11-120
MB_LEN_MAX macro 11-99
mbstowcs function 11-120
mbtowc function 11-120
memchr function 11-123
memcmp function 11-122
memcpy function 11-121
memmove function 11-121
memory management functions 11-118
memset function 11-123
messages
  compiler 2-23
  diagnostic 2-23
miscellaneous functions 11-123
misplaced_lint_directive
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-247, G-278
  lint error message example F-247, G-278
mixed-mode
  example 3-30
mktime function 11-125
modf function 11-103
multibyte
  character functions 11-120
  characters E-168, E-176
  string functions 11-120
multiple mode compiling 3-29
MVBITS man page F-220

N

name space
  ordinary identifiers F-200, F-264, G-299
  struct, union, and enum F-263, G-298
  tags F-272, G-308
NAME_MAX macro 11-100
native format
  double-precision A-141, A-149
  single-precision A-140, A-148
negative_to_uns
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-248, G-279
neg_shift
  compatibility mode actions F-196, G-228
  description F-193, G-225
  error message example F-265, G-300
  lint error message example F-265, G-300
next_task pragma B-151
no_arg_type
  compatibility mode actions F-197, G-228
  description F-193, G-225

error message example F-228, G-258
no_external_declaration
compatibility mode actions F-197, G-228
description F-193, G-225
error message example F-275, G-310
no_parallel pragma B-151, B-154
no_recurrence pragma B-152, B-153
no_side_effects pragma B-154
no_vector pragma B-154
__NO_INLINE macro 9-82
__NO_INLINE_BINT macro 9-82
__NO_INLINE_CTYPE macro 9-82
__NO_INLINE_MATH 9-79
__NO_INLINE_MATH macro 9-82
__NO_INLINE_SIGNAL macro 9-82
__NO_INLINE_STDIO macro 9-82
__NO_INLINE_STDLIB macro 9-82
__NO_INLINE_STRING macro 9-82
__NO_INLINE_TIME macro 9-82
non_int_bit_field
compatibility mode actions F-197, G-228
description F-193, G-225
error message example F-207, G-234
note
converting error messages F-192, G-224
converting errors to warnings F-192, G-224
COVUEshell requires .C file extensions 2-8
-d option F-192, G-224
do not directly call intrinsic functions 9-80
IEEE 754 specifications A-139, A-147
long float is a CONVEX extension A-141, A-149
long long int is a CONVEX extension A-137, A-145
object files of x.s and x.a have the same name 2-14
nothing_declared
compatibility mode actions F-197, G-228
description F-194, G-226
error message example F-251, G-282
null characters
appending to stream E-176, E-184
NULL macro
expansion E-172, E-180
use 11-110, 11-111, 11-117, 11-121, 11-124
null_effect_expression
compatibility mode actions F-197, G-228
description F-194, G-226
error message example F-223, G-252

O

object file compatibility 2-25
object pointers, converting 3-34
octal constants 3-35
octal digits (8 and 9) 3-34
offsetof macro 11-110
operation
bitwise, on a signed integer E-169, E-177
bitwise, on an unsigned integer E-169, E-177
right shift, negative signed integer E-169, E-177
operations on files 11-112
operators
old-style (=+), etc. 3-36
pasting (##) 12-130
stringizing (#) 12-130
optimization
basic-block 7-61
function-level 7-62
machine-dependent 7-60
options 7-67
parallel 7-60, 7-64
pragmas 7-67
report 2-23, 7-65
scalar 7-59, 7-60
user-directed 7-67
vector 7-60, 7-63
options
-alias array_args 2-17
-alias cautious 2-17
-alias ptr_args 2-17
-alias standard 2-18
-alias worst 2-18
-asm 2-12
-B 2-20
-C 2-11
-c 2-12
-cxdb 2-16
-D 2-11
-d 2-15, F-191, G-223
-db 2-16
-ds 2-18
-E 2-11
-ep 2-18, B-159
-ext 2-10
-extern distinct 2-13
-extern same 2-13
-fd 2-13
-fi 2-13
-float dp_const 2-13

-float dp_ops 2-13
-float sp_const 2-13
-float sp_ops 2-13
-fn 2-13
-I 2-12
-k 2-12
-no 2-18
-nw 2-15
-O 2-18
-o 2-20
-O0 2-18
-O1 2-18
-O2 2-18
-O3 2-18
-or 2-15
-P 2-12, 2-16
-pa 2-17
-pab 2-17
-par 2-17
-parens explicit 2-13
-parens ignore 2-13
-parens implicit 2-13
-pb 2-16
-pcc 2-10
-pg 2-16
-re 2-14
-rl 2-18
-S 2-14
-sc 2-15
-std 2-10, 3-29
-str 2-10
-string read_only 2-14
-string read_write 2-14
-tl 2-20
-tm 2-14
-tm C1 2-14
-tm c1 2-14
-tm C2 2-14
-tm c2 2-14
-U 2-12
-uo 2-18
-ur 2-18
-va 2-18
-vn 2-20
options, general
code generation 2-12
compatibility 2-9
compatibility with other compilers 2-20
diagnostic 2-15
introduction 1-2
miscellaneous 2-20
optimization 2-17
preprocessor 2-11
utility support 2-15
ordering documentation
how to xv

P

-p option 6-56
padding
data types E-171, E-179
structures A-145, A-153
parallel strip mining B-155
parallel optimization 7-60, 7-64
description 7-60, 7-64
inhibitors 7-64
parameter passing to FORTRAN 8-75
_parity
function description D-164, D-172
man page F-217
pasting operator
/**/ 3-36
12-130
-pb option 6-56
_pbit
function description D-164, D-172
man page F-216
_pbits
function description D-165, D-173
man page F-216
pcc command line 3-30
pclose function 11-116
perror function 11-116
-pg option 6-56
pmd utility 5-50
pointer_alignment_efficiency
compatibility mode actions F-197, G-228
description F-194, G-226
error message example F-257, G-289
lint error message example F-257, G-289
pointer
data type A-142, A-150
generic A-143, A-151
pointers
converting function pointers 3-34
converting object pointers 3-34
to arrays A-146, A-154
_popcnt
function description D-165, D-173
man page F-217
popen function 11-116
_poppar
function description D-165, D-173
man pageF-217

porting to backward-compatible mode 3-30
POSIX
  conforming applications 3-28
  definition 2-10, 3-27, 11-91
POSIX Extensions
  limits.h 11-99
  setjmp.h 11-106
  signal.h 11-107
  stdio.h 11-116
  time.h 11-126
POSIX functions
  close 3-37
  creat 3-37
  fseek 3-37
  open 3-37
  read 3-37
  unlink 3-37
  write 3-37
_POSIX_ARG_MAX macro 11-99
_POSIX_CHILD_MAX macro 11-99
_POSIX_LINK_MAX macro 11-99
_POSIX_MAX_CANON macro 11-99
_POSIX_MAX_INPUT macro 11-99
_POSIX_NAME_MAX macro 11-99
_POSIX_NGROUPS_MAX macro 11-100
_POSIX_OPEN_MAX macro 11-100
_POSIX_PATH_MAX macro 11-100
_POSIX_PIP_BUF macro 11-100
_POSIX_SOURCE macro 2-21, 3-28, 3-29
postmortem dump 5-50
pow function
  description 11-103
  returns pow(|x|) on domain error E-173, E-181
pp_argcount
  compatibility mode actions F-197, G-228
  description F-194, G-226
  error message example F-285, G-321
pp_argsended
  compatibility mode actions F-197, G-228
  description F-194, G-226
pp_badflag
  error message example F-232, F-280, G-262, G-316
pp_badkfile
  error message example F-247, G-279
pp_badstr
  compatibility mode actions F-197, G-228
  description F-194, G-226
  error message example F-240, G-270
pp_badtp
  compatibility mode actions F-197, G-228
  description F-194, G-226
  error message example F-240, G-270
pp_extra
  compatibility mode actions F-197, G-228
  description F-194, G-226
  error message example F-225, G-254
pp_idexpected
  compatibility mode actions F-197, G-228
  description F-194, G-226
pp_line_range
  compatibility mode actions F-197, G-228
  description F-194, G-226
pp_macro_arg
  compatibility mode actions F-197, G-228
  description F-194, G-226
  error message example F-245, G-275
pp_macro_redefinition
  compatibility mode actions F-197, G-229
  description F-194, G-226
  error message example F-245, G-276
pp_macro_redefinition_cmdl
  compatibility mode actions G-229
  description F-194, G-226
pp_malformed_directive
  compatibility mode actions F-197, G-229
  description F-194, G-226
pp_old_dir
  compatibility mode actions F-197, G-229
  description F-194, G-226
  error message example F-252, G-284
pp_parse
  compatibility mode actions F-197, G-229
  description F-194, G-226
pp_undef
  compatibility mode actions F-197, G-229
  description F-194, G-226
pp_undef_cmdl
  compatibility mode actions F-197, G-229
  description F-194, G-226
pp_unrecognized_directive
  compatibility mode actions F-197, G-229
  description F-195, G-227
pp_unrecognized_pragma
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-260, G-292
#pragma, causes no actions E-172, E-180
pragma B-149, B-157
  begin_tasks B-151
  end_tasks B-151
  force_parallel B-151
  force_parallel_ext B-151, B-152
  force_vector B-152
  max_trips B-153

next_task B-151
no_parallel B-151, B-154
no_recurrence B-152, B-153
no_side_effects B-154
no_vector B-154
optimization 7-67
prefer_parallel B-155
prefer_parallel_ext B-155
prefer_vector B-155
pstrip B-155, B-156
restrictions B-150
scalar B-151, B-152, B-156
select B-157
synch_parallel B-158
unroll B-158
vstrip B-159
precision
float operations 3-36
floating-point operations 3-36
predefined macro names 3-34
predefined symbols 2-20
prefer_parallel pragma B-155
prefer_parallel_ext pragma B-155
prefer_vector pragma B-155
preprocessor
description of 12-127
operators 12-130
preprocessor directive
#define 12-127
#elif 12-131
#else 12-131
#endif 12-131
#error 12-132
#if 12-131
#ifdef 12-131
#ifndef 12-131
#line 12-132
#pragma 12-132
#undef 12-129
printf function 11-113
processor status word 8-70
product reduction operator
floating-point B-156
profiler, CXpa 6-57
program counter 8-70
program input and output 10-86
promotion, function parameter 3-36
pseudo-random sequence generation functions 11-118
pstrip pragma B-155, B-156
ptrdiff_t type 11-110, E-170, E-178
putc function 11-114
putchar function 11-114
puts function 11-114

Q

qsort function 11-119

R

raise function 11-106
rand function
return value 11-118
with RAND_MAX 11-117
RAND_MAX macro 11-117
range of values, char E-169, E-177
rcvtir function 11-105
realloc function
description 11-118
request size zero E-180, E-188
recurrence B-153
apparent B-153
real B-153
reentrant code, use with -re 7-64
register storage class
impact E-170, E-178
legal uses F-282, G-318
remainder, sign in integer division E-169, E-177
remove function
description 11-112
execution on open file E-176, E-184
rename function
action on an existing file E-176, E-184
description 11-112
reordering expressions 3-35
report, optimization 2-23
reporting problems xv
representation
char A-136, A-144
character data A-147, A-155
double A-141, A-149
enum A-138, A-146
float A-140, A-148
int A-136, A-144
long A-136, A-144
long double A-141, A-149
long int A-136, A-144
long long A-137, A-145
long long int A-137, A-145
pointer A-142, A-150
short A-136, A-144
short int A-136, A-144

string A-146, A-147, A-154, A-155
restrictions, pragma use B-150
rewind function 11-115
right shift of a negative integer E-169, E-177
rindex function 11-124
runtime
functions, calling 11-93
library 11-91
messages 2-24
stack 8-69

S

scalar optimization 7-59, 7-60
scalar pragma B-151, B-152, B-156
scanf function 11-113
SCHAR_MAX macro 11-99
SCHAR_MIN macro 11-99
scope, external declarations 3-35
search functions 11-123
searching and sorting functions 11-119
SEEK_CUR macro 11-111
SEEK_END macro 11-111
SEEK_SET macro 11-111
select pragma B-157
set_but_not_used
compatibility mode actions F-197, G-229
description F-195, G-227
error message example F-265, G-300
lint error message example F-265, G-300
setbuf function 11-112
setjmp function
description 11-105
optimization with F-255, G-287
_setjmp function
optimization with F-255, G-287
setjmp.h contents 11-105
setlocale function 11-101
setvbuf function
associated macros 11-110
description 11-112
shift_too_large
compatibility mode actions F-197, G-229
description F-195, G-227
error message example F-266, G-301
lint error message example F-266, G-301
short
alignment E-171, E-179
data type A-135, A-143
short int data type A-135, A-143
short_cvt_truncates
compatibility mode actions F-197, G-229
description F-195, G-227
error message example F-215, G-243
SHRT_MAX macro 11-99
SHRT_MIN macro 11-99
sig_atomic_t type 11-106
SIG_BLOCK macro 11-108
SIG_CATCH 3-37
SIG_HOLD 3-37
SIG_SETMASK macro 11-108
SIG_UNBLOCK macro 11-108
sigaction
function 11-108
structure 11-107
sigaddset function 11-108
SIGCLD macro 3-37
sigcontext structure 11-107
sigdelset function 11-108
sigemptyset function 11-108
sigfillset function 11-108
SIGILL macro, reset of default handling E-175, E-183
sigismember function 11-108
sigjmp_buf type 11-106
siglongjmp function 11-106
sigmask macro 11-107
signal function 11-106
signal handler functions 3-37
signal.h
ANSI C changes 3-37
contents 11-106
future changes 3-38
signals
blocking of E-175, E-183
default actions E-175, E-183
semantics E-174, E-182
SIGILL, reset of default handling E-175, E-183
what signals are available E-174, E-182
SIGNALS_IN_PROG function 3-37
sigpending function 11-109
sigprocmask
associated macros 11-108
function 11-109
sigsetjmp function
description 11-106
optimization with F-255, G-287
sigsuspend function 11-109
sigvec structure 11-107
sin function
description 11-103
returns sin(0) on domain error E-173, E-181
single mode compiling 3-28

single-precision
IEEE format A-140, A-148
native format A-140, A-148
sinh function
description 11-103
returns -HUGE_VAL on domain error E-173, E-181
size_t type 11-110, 11-111, 11-117, 11-121, 11-124
sizeof operator
function as argument F-266, G-302
void as argument F-267, G-302
skip_to_char
compatibility mode actions F-197, G-229
description F-195, G-227
error message example F-272, G-307
skip_to_eof
compatibility mode actions F-197, G-229
description F-195, G-227
error message example F-272, G-307
space characters
elimination text stream file E-176, E-184
sprintf function 11-113
sqrt function
description 11-103
returns sqrt(|x|) on domain error E-173, E-181
srand function 11-118
sscanf function 11-113
stack frame 8-70
stack pointer 8-69
static storage class
changing from extern to static F-201, G-297
changing to static from extern F-198, F-263, G-294, G-297
stdarg.h contents 11-109
__STDC__ macro 2-21
__stdc__ macro 2-21
stddef.h contents 11-110
stderr device 11-111
stdin device 11-111
stdio.h
ANSI C changes 3-37
contents 11-110
future changes 3-38
stdlib.h
contents 11-117
future changes 3-38
stdout device 11-111
storage class
auto F-282, G-318
function F-235, G-265
function parameter F-236, G-266
register E-170, E-178, F-282, G-318
strcat function 11-121
strchr function 11-123
strcmp function 11-122
strcoll function 11-100, 11-122
strcpy function 11-121
strcspn function 11-123
strerror function
description 11-123
error message strings E-180, E-189
strftime function
description 11-126
LC_TIME 11-101
strict compatibility mode
description 3-27
example 3-29
strict_syntax
compatibility mode actions F-197, G-229
description F-195, G-227
error message example F-250, G-281
string concatenation operator (#) 12-130
string conversion functions 11-117
string handling functions 11-121
string literals, identical 3-35
string.h
contents 11-121
future changes 3-38
stringizing operator (#) 12-130
strings.h, ANSI C changes 3-37
strip mining B-152
parallel B-155
select pragma B-157
strip-mine length, pstrip pragma B-155
strlen function 11-124
strncat function 11-122
strncmp function 11-122
strncpy function 11-121
strpbrk function 11-123
strrchr function 11-123
strspn function 11-123
strstr function 11-123
strtod function 11-118
strtok function 11-123
strtol function 11-118
strtoul function 11-118
struct
alignment E-171, E-179
assignments F-236, G-266
data type A-144, A-152
empty declaration 3-36
empty declarations 3-34
representation A-144, A-152

structure
  alignment E-171, E-179
  member alignment A-144, A-152
  padding A-145, A-153
strxfrm function 11-100, 11-122
sum reduction operator
  floating-point B-156
switch case label, data type F-270, G-306
switch control expression, data type F-271, G-306
symbolic debugger 5-50
synch_parallel pragma B-158
system function
  description 11-119
  valid argument E-180, E-189
system functions and Stream functions 10-85

T

table
  available signals and their semantics E-182
  character constant representation E-168, E-176
  characters checked by ctype.h functions E-173, E-181
  compatibility modes 3-27, 11-91
  compiler directives C-161, C-170
  compiler options for profiling 6-56
  default actions for signals E-175, E-183
  diagnostic condition default settings F-196, F-197, G-228, G-229
  errno values of fgetpos and ftell E-177, E-185
  errno values of fgetpos, fsetpos, and ftell 11-115
  error messages E-178, E-186
  floating-point bit length A-139, A-147
  FORTRAN and C declarations 8-75
  integer conversion E-177
  integer conversions E-169
  integral ranges A-136, A-144
  integral type bit length A-135, A-143
  ipc/network argument error messages E-179, E-187
  long float range: native and IEEE A-150
  long long data type range A-137, A-145
  math error messages E-177, E-187
  math function return values 11-104, E-173, E-181
  native and IEEE floating-point ranges A-139, A-147
  native and IEEE long float range A-142
  nfs error messages E-177, E-187
  nonblocking and interrupt I/O error messages E-177, E-187
  optimization pragmas 7-68
  restrictions on pragma use B-150, B-158
  signals and semantics E-174
  SystemV record locking error messages E-177, E-188
  tape system error messages E-180, E-188
  trigraph representations 3-35
  type of postmortem dump contents 5-51
TAC (Technical Assistance Center) xv
tan function 11-103
tanh function 11-104
tasking directives
  maximum number F-231, G-261
tasking pragmas (directives)
  example F-207, G-234
  maximum number F-231, G-261
technical assistance center (TAC) xv
technical assistance center
  telephone number for xv
technical assistance
  obtaining xv
text stream
  terminating newline character E-176, E-184
  truncation E-176, E-184
time
  conversion functions 11-126
  functions 11-124
  manipulation functions 11-125
time function 11-125
__TIME__ macro 2-21, E-172, E-180
time_t type 11-124
tm structure 11-125
TMP_MAX macro 11-111
tmpfile function 11-112
tmpnam function
  description 11-112
  L_tmpnam 11-111
  TMP_MAX 11-111
toascii
  function 3-37
  macro 11-95
tolower function 11-94
_tolower
  function 3-37
  macro 11-95
toupper function 11-95
_toupper
  function 3-37
  macro 11-95

trigraphs 3-35
trouble reports xv
truncation of text streams E-176, E-184
type conversion from FORTRAN 8-74
typedef names in formal parameters 3-34
tzname variable 11-126

U

UCHAR_MAX macro 11-99
UINT_MAX macro 11-99
unary minus, operands F-253, G-285
unary plus, operands F-253, G-285
#undef, use of 11-92
ungetc function 11-114
union
  alignment E-171, E-179
  assignments F-236, G-266
  data type A-143, A-151
  empty declarations 3-34
  order of accessing members E-170, E-178
__unix__ macro 2-21
unix macro 2-21
unroll pragma B-158
uns_compare_neg
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-286, G-323
uns_compare_zero
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-286, G-322
unsigned_suffix
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-241, G-271
user-directed optimization 7-67
USHRT_MAX macro 11-99

V

va_arg macro 11-109
va_end macro 11-109
va_list type 11-109
va_start macro 11-109
varargs_on_proto
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-281, G-317
  lint error message example F-281, G-317
varargs_too_large
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-282, G-318
  lint error message example F-282, G-318
vector length B-159
vector, optimization 7-60, 7-63
vectorization
  description 7-60, 7-63
  inhibitors 7-64
vfprintf function 11-113
void data type
  description A-143, A-151
  typical uses F-284, G-320
void_pointer_cast
  compatibility mode actions F-197, G-229
  description F-195, G-227
  error message example F-259, G-291
  lint error message example F-259, G-291
volatile
  what is an access E-171, E-179
vprintf function 11-113
vsprintf function 11-113
vstrip pragma B-159

W

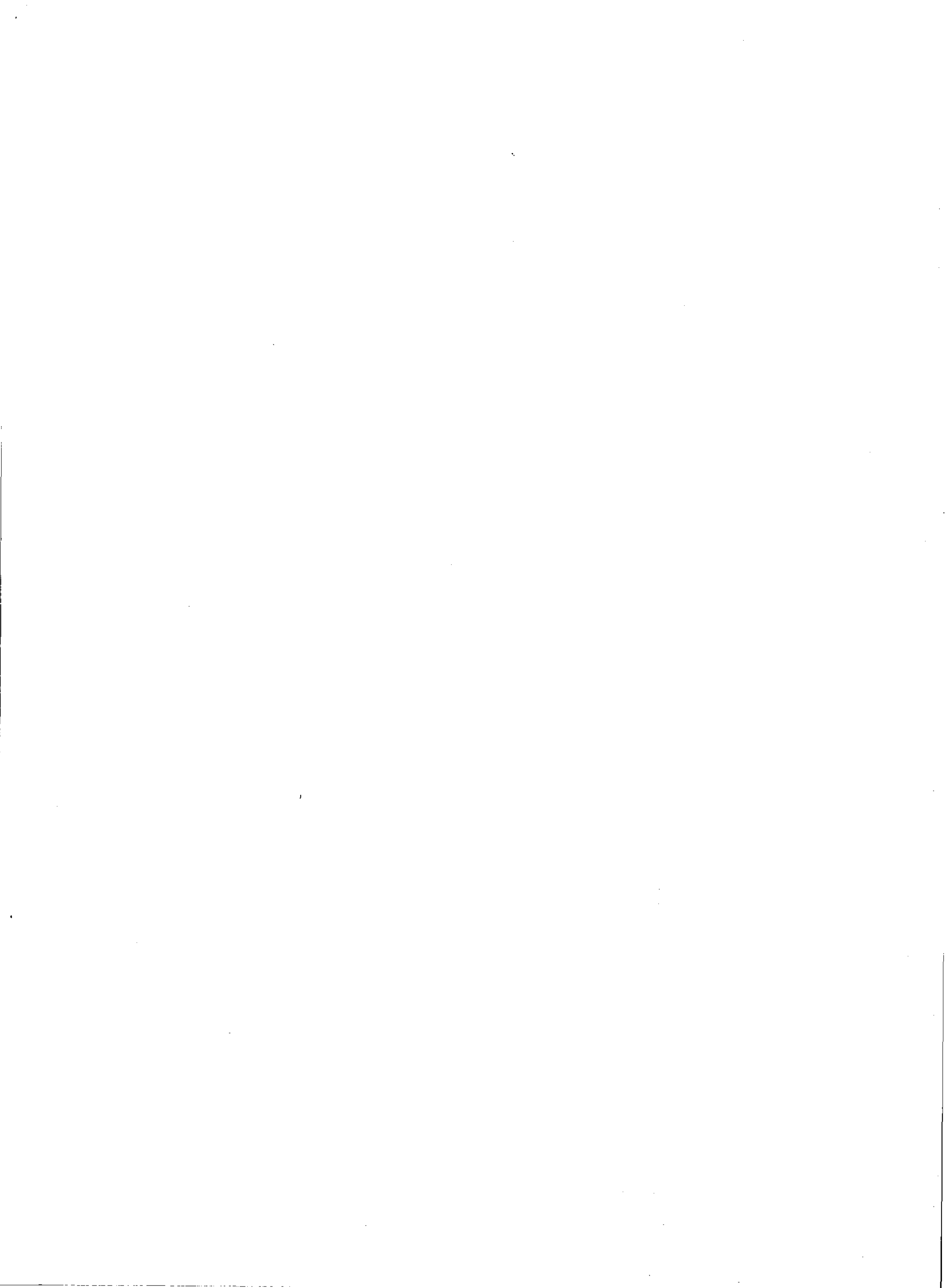
wchar_t type 11-110, 11-117
wcstombs function 11-120
wctomb function 11-120
wide character constant, definition of F-241, G-272

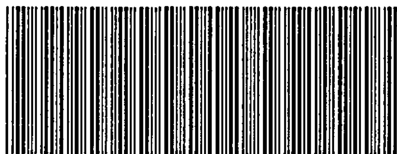
Y

y0 function 11-105
y1 function 11-105
yn function 11-105

Z

zero-length arrays 3-34
zero-lenth files E-176, E-184





Order Number
DSW-086

Document Number
720-000630-205