CONVEX C User's Guide

Document No. 720-000630-203

Fourth Edition
April 1990

CONVEX Computer Corporation
Richardson, Texas USA

CONVEX C User's Guide
Order No. DSW-086
Fourth Edition

© 1986, 1987, 1988, 1989, 1990 CONVEX Computer Corporation
All rights reserved.

This document is copyrighted. This document may not, in whole or part, be copied, duplicated, reproduced, translated, stored electronically, 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 (CONVEX) 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 WARRANTY 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 OR ANY THIRD PARTY.

CONVEX and the CONVEX logo ("C") are registered trademarks of CONVEX Computer Corporation.
ConvexOS is a trademark of CONVEX Computer Corporation.
UNIX is a registered trademark of AT&T Bell Laboratories.

Printed in the United States of America

Revision Information for *CONVEX C User's Guide*

Edition	Document No.	Description
Fourth	720-000630-203	Released with CONVEX C software V4.0, April 1990. Chapters on Data Types, Calling Conventions, and Runtime Libraries moved to *CONVEX C Language Reference Manual.* Appendices on Data Representations, the Preprocessor, and Compiler Directives moved to *CONVEX C Language Reference Manual.* Appendix on Runtime Libraries deleted. Appendix on Compiler messages expanded to include an explanation for each message. Chapters on Program Development Tools and Profiling Tools added.
Third	720-000630-201	Released with CONVEX C software V3.0, January 1989.
2.0	720-000630-200	Released with Vector C software V2.0, January 1988.
1.0	720-000430-000	Released with Vector C software V1.0, August 1986.

Table of Contents

1 Compiling a Simple Program
What is a Compiler? 1-1
Creating a Source File 1-1
Compiling One Source File 1-2
Compiling More Than One Source File 1-2
Compiler and Linker 1-4
Libraries 1-5

2 Compiler Fundamentals
Introduction 2-1
Compiler Features 2-1
File-Naming Conventions 2-2
Compilation Process 2-2
Preprocessor Options 2-3
Code Generation Options 2-4
Diagnostic Options 2-6
Compatibility Options 2-6
Debugging and Profiling Options 2-7
Optimization Options 2-9
Miscellaneous Options 2-11
Compatibility With Options of Other Compilers 2-11
Predefined Symbols 2-12
Linker Usage 2-12
Environment Variables 2-13
Compiler Messages 2-14
Mixed Compatibility Modes 2-15
Object File Compatibility 2-15
Notes and Cautions 2-16

3 Program Development Tools
lint Utility 3-1
make Utility 3-3
indent Utility 3-3
error Utility 3-4

4 Debugging Programs
Introduction 4-1
CONVEX Consultant Debuggers 4-1
Cross-Reference Generator 4-3
Assembly-Language Debugger 4-4

5 Profiling Programs
Introduction 5-1
CONVEX Consultant Profilers 5-1
CXpa Profiler 5-2

6 Optimizing C Programs
Introduction 6-1
Scalar Optimization 6-2
Vector Optimization 6-4
Parallel Optimization 6-5
The Optimization Report 6-6
User-Directed Optimization 6-7

Appendices

A **Error Messages** A-1
Error Message Control A-1
Error Message Catalog A-4
B **Reporting Problems** B-1
Technical Assistance Center B-1
The contact Utility B-1
Prerequisites B-1
Tips on Using the contact Utility B-3
Using the contact Utility B-4

List of Tables

2-1 File Extension Conventions 2-2
2-2 Compatibility Modes 2-6
2-3 Option Compatibility 2-11
4-1 Postmortem Dump Contents 4-2
5-1 Compiler Options for Profiling 5-2
6-1 Optimization Options 6-7
6-2 Optimization Directives 6-9
6-3 Restrictions on Directive Use 6-10
A-1 Diagnostic Condition Default Settings A-3
A-2 Index to Diagnostic Messages A-3

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 Multiple Source Files 1-3
1-5 Compiler and Linker Interactions 1-4
1-6 Linking Library Routines 1-5
2-1 Compilation Process 2-2
2-2 Options Required for Debuggers and Profilers 2-8
2-3 Object File Compatibility 2-15

Preface

Purpose and Audience

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 the previous version of CONVEX C. This document is intended for use by novice programmers as well as more experienced programmers.

Organization

This manual 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, "Program Development Tools," describes some of the tools that assist in program development.
- Chapter 4, "Debugging Programs," describes tools that aid in finding errors in a program.
- Chapter 5, "Profiling Programs," describes tools that are used to find time-consuming parts of a program.
- Chapter 6, "Optimizing C Programs," describes basic techniques that the compiler uses to optimize a program.
- Appendix A lists the C compiler error messages and a short explanation of each message.
- Appendix B provides instructions for reporting software or documentation problems to the CONVEX Technical Assistance Center (TAC).

Notational Conventions

The following conventions are used in this document:

- Words enclosed in rounded rectangles are keyboard keys that you press. For example, (RETURN) denotes the carriage return key. Words separated by a hyphen and enclosed in rounded rectangles indicate two keys that you must press simultaneously. For example, (CTRL-X) indicates that you must press the (CTRL) key while simultaneously pressing the keyboard (X) character key.
- 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.
- *Italics* designate user-suplied variables in a command-line example, introduce new terms of great importance, identify variables in mathematical equations, and indicate titles of documents.
- `Constant-width font` is used for input and output. This includes: command names and options, system calls, data structures and types, directives, program statements, display examples, printout examples, and error messages returned.
- **Bold font** is used to clearly identify 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 repitition of the preceding item(s).

    Consider the following example:

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

    where `COMMAND` must be typed as it appears; *input_file* indicates a file name that must be supplied by you; the horizontal ellipsis in brackets indicates that additional input file names may be supplied; either `a` or `b` must be supplied; and *output_file* indicates an optional file name.
- References to the *ConvexOS Programmer's Reference* appear in the form adb(1), where the name of the man page is followed by its section number enclosed in parentheses.

Associated Documents

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

The following documents are available in the document set that contains this document:

- *CONVEX C Language Reference Manual* describes some topics of the C language. One chapter describes ANSI C features that are specific to the CONVEX compiler, and another chapter details the differences between the Common C compiler and the backward-compatible mode of CONVEX C V4.0.
- *CONVEX C Optimization Guide* presents a step-by-step method of program optimization. Background information is included and forms a foundation for concepts presented throughout the document.
- *CONVEX C Quick Reference* provides quick access to function prototypes, compiler directives, compiler options, and language features.
- *ANSI C Concepts* is a high-level introduction to the ANSI C standardized programming language. It also explains the best approach to port applications to CONVEX C V4.0.
- *CONVEX C Programmer's Reference* contains man pages for all the ANSI C functions and the C compiler.
- *CONVEX C Master Index* contains an index for all the C documentation.

The following documents are provided by CONVEX Computer Corporation to help you with ConvexOS:

- *CONVEX UNIX Primer* provides an introduction for users who have not previously used the ConvexOS operating system.
- *ConvexOS Programmer's Manual,* Parts I and II, is the standard reference for the ConvexOS operating system.
- *CONVEX UNIX System Manager's Guide* contains information needed to manage and administer a CONVEX supercomputer system.
- *CONVEX UNIX Tutorial Papers* 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.
- *CONVEX UNIX Utilities User's Guide* provides detailed information on the CONVEX Assembler, Loader, text editors, and `adb` debugger.
- *CONVEX Performance Analyzer (CXpa) User's Guide* provides information required to use the CONVEX Performance Analyzer tool, an optional product.
- *CONVEX Assembly-Language User's Guide* provides information required to use the CONVEX Assembler.
- *CONVEX Loader User's Guide* provides information required to use the CONVEX link editor.

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.

Ordering Documentation

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

In some situations, the current edition may not be needed. 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):

Within the continental U.S.	1(800)952-0379.
From locations in Alaska, Hawaii, and Canada	1(214)497-4379.
From all other locations	contact the nearest CONVEX office.

Reader's Forum

If you want to mail your comments to us, please use the form at the end of this manual and list the document page number with your questions and comments.

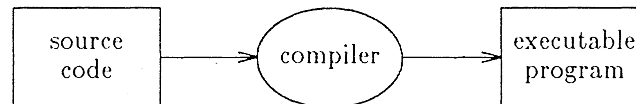
Chapter 1
Compiling a Simple Program

This chapter is a brief introduction to compilation. Several small examples are provided. Terms used later in this manual are introduced and some basic capabilities of the compiler are presented. 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 Editor User's Guide* for information on text editors.

What is a Compiler?

A compiler translates source code that people can understand into an executable program that a computer can understand. 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 print the line `A simple program`, use a text editor to enter the code as shown below:

```
#include <stdio.h>

main()
{
        (void) puts("A simple program");
}
```

Name the file myprog.c, save it, and exit from your editor.

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 have this extension. (The COVUEshell environment requires ".C".) The compiler does not 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 determine its behavior; The features are selected by including options on the command line of the compiler. The format of an option consists of a hyphen followed immediately by a letter. The letter chosen depends on the option. Some options require arguments. The syntax for an option varies; some require a space to delimit the argument, while others do not.

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 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 may be divided into source files containing related functions. For example, all input functions may be grouped in one file.

Fortunately, the compiler can accommodate such situations. 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

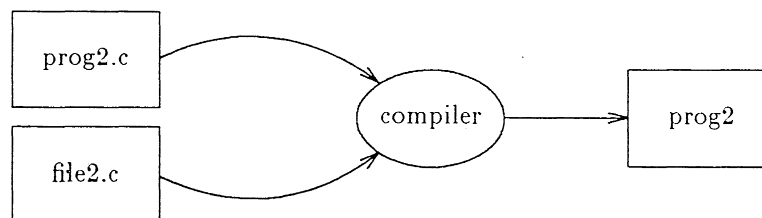
```
#include <stdio.h>

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 prog.c and `second_line` is defined in file2.c. The actions of the compiler are illustrated in Figure 1-4.

Figure 1–4: Multiple Source Files



The command to create the executable program prog2 is

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

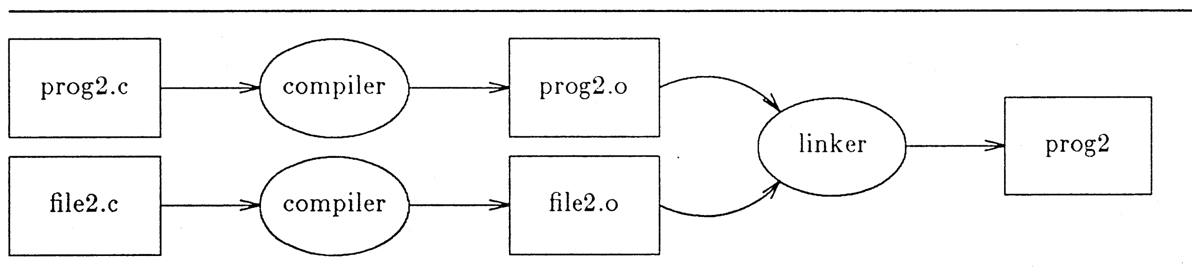
After the program is created, run it by entering **prog2**. The compiler compiles the source files specified on the command line.

Compiler and Linker

The compiler does not create an executable program in one step. First, it creates an intermediate file that contains machine code. Most of the machine code is a translation of the source file, but some of it refers to source code in other files. For example, 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 references and the 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.

After the compiler has compiled all C source files listed on its 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 all the object files generated by the compiler, and the output from the linker is the executable program. The process used to create the previous 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.

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, then 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. Lastly, if prog2.c is modified but file2.c is not, then 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
```

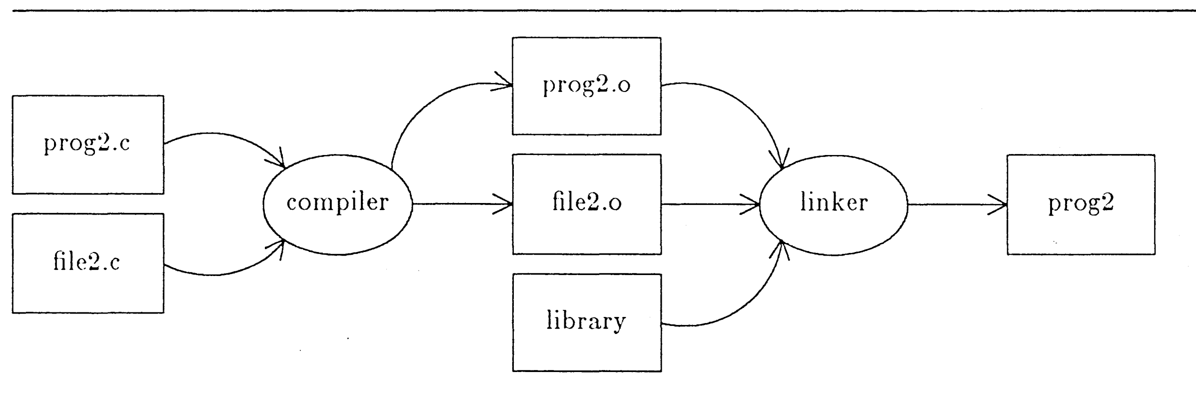
compiles prog3.c and then links prog3.o, file3.o, and file4.o.

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 may be collections of functions that you create for your programs. Each of the functions in the libraries may 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 the process of compilation.

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 instance, 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
```

Chapter 2
Compiler Fundamentals

Introduction

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

This chapter discusses options that can be selected 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 which was bundled with CONVEX systems. This compiler and the previous release of CONVEX C are no longer supported.

Compiler Features

The current release of the CONVEX C compiler is ANSI C conforming, as specified in the ANSI X3J11/90-013 document. Programs written for the ANSI C compatible 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.
- Is compliant with IEEE Std 1003.1-1988 (POSIX).

The next two sections discuss material that is necessary to understand the behavior of the compiler.

File-Naming Conventions

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

Table 2-1: File Extension Conventions

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

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.

Compilation Process

The C compiler is invoked by entering the following command line:

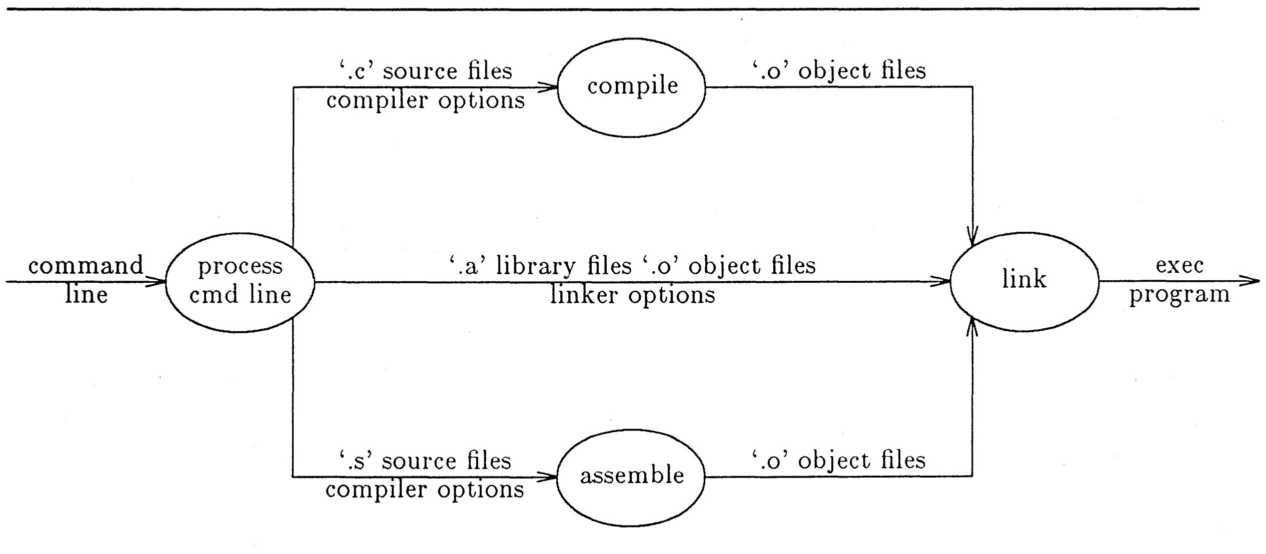
cc [*options*] *files* [*linker-options*]

where

options	is zero or more of the options specified in the following sections.
files	represents one or more source files to be compiled, object files to be linked, assembly-language files to be assembled, or libraries to be linked.
linker-options	is zero or more of the linker (`ld(1)`) options as described in the *CONVEX Loader User's Guide*.

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



The linker receives options from the command line, but the assembler is dependent on the options that the C compiler uses. Options accepted by the assembler are common to the CONVEX C compiler and the CONVEX assembler. Description of assembler options can be found in the *CONVEX Assembly-Language User's Guide.*

Compiler options may be categorized as:

- Preprocessor options.
- Code generation options.
- Diagnostic options.
- Compatibility options.
- Debugging and Profiling options.
- Optimization options.
- Miscellaneous options.

Each of these categories is discussed in the remainder of this chapter.

Preprocessor Options

The preprocessor is a program that the compiler uses to process a source file before it is compiled. Common uses of the preprocessor include the definition of:

- Manifest constants.
- Conditionally compiled source code.
- Function-like macros.

Several options are used to control the preprocessor. For more information on the preprocessor, consult the *CONVEX C Language Reference Manual.*

Preprocessor options 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 may be specified; they are searched in the order specified on the command line.
`-k`	Runs the preprocessor on the named C source files and generate dependency descriptions for `make`(1); results are printed out on the standard output stream. All file names on the command line that are not C source files are ignored.
-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. To locate errors in a program, it is convenient to include debugging code in a program. 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 the 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.

Some programming projects may be large. When there are many source files and header files, relationships between the files may 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 are discussed in Chapter 4, "Debugging Programs."

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 numbers in the program. The code generation options are:

-asm — This flag is silently ignored; it is no longer required.

-c — Suppresses the linking phase of compilation. The object module that is generated from the file 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 two different files declare an uninitialized variable (e.g., int i;) at file scope. 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; 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 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 may 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 may run faster, but the results may not be as precise as 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 explicit`
`-parens ignore`
`-parens implicit` This option affects the interpretation of parentheses used in floating-point operations. 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. This is the default in the backward-compatible and extended modes. If you specify the `implicit` argument, the compiler honors all parentheses, as well as grammar and associativity rules, and 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`, is to generate nonreentrant 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 want to call that function from a parallel region. It is always safe to use `-re`; the text segment is 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 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* may be c1, C1, c2, or C2; the default value is the type of machine hosting the compiler. 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 machine on a C1 machine. The following 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.

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, no source files for both assembly language and C should have the same file name stem (file name without the extension).

Diagnostic Options

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

-d name[={w|e}] Suppresses error or warning messages represented by *name*. This option can be used to convert a warning message to an error message and vice versa. The specific warning and error messages are listed in Appendix A, "Error Messages".

`-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.

Compatibility Options

Recently there have been many attempts to standardize the environments in which computers operate. These environments include operating systems and programming languages. This release of the CONVEX C compiler coincides with both the POSIX operating systems standard and the ANSI C language standard. This compiler can compile source code that conforms to one or both standards.

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. Similarly, the strict mode is used for programs that must be portable to other systems.

The options used to select the compatibility mode are:

- `-ext` Selects the extended mode. This mode provides an extended implementation of ANSI C and an ANSI C and POSIX compatible library system. 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. *CONVEX ANSI C Concepts* 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.

If the conforming mode is needed, the _POSIX_SOURCE manifest constant must be defined on the command line using the `-D` compiler option or in the source files using the `#define` preprocessor directive.

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

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

To create a program that is POSIX conforming, the constant _POSIX_SOURCE must be defined on the first line of each source file and compiled with the 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 then the program would not have been POSIX conforming.

The following line creates a program for the extended mode:

```
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, the following command line is used:

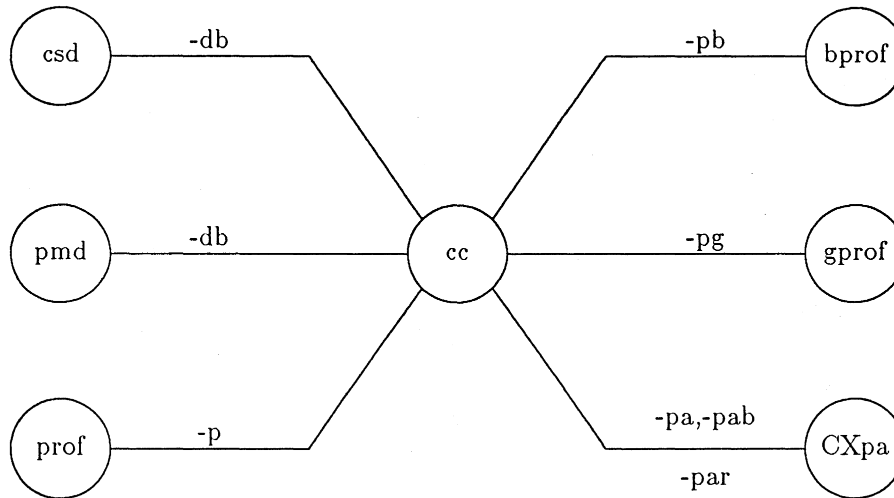
```
cc -pcc file.c
```

The `-pcc` option is not completely compatible with the Common C compiler. Refer to *CONVEX ANSI C Concepts* for information about the differences.

Debugging and Profiling Options

Several options provide support for program development tools. For example, 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:

`-db` Produces additional information for use by the symbolic debugger, `csd(1)`, and the post-mortem dump utility, `pmd(1)`. 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 may be reordered. The `-no` compiler option prevents this reordering. Also, 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. An execution profile can then be generated using `prof(1)`.

`-pb` Includes instrumentation that produces an execution profile named bmon.out at normal termination. Listings of source-level execution counts can then be obtained using `bprof(1)`.

`-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. An execution profile can then be generated using `gprof(1)`.

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

`-par` Includes instrumentation for routine, loop, and region level profiles using the CXpa utility. Refer to the *CONVEX Performance Analyzer (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 6, "Optimizing Programs." For more specific information, such as optimizing a program using optimization directives, refer to the *CONVEX C Optimization Guide*.

It may not be possible to use all of the optimization options. There are two versions of CONVEX C: a scalar optimizing version bundled with each machine 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 ignores associated compiler options.

`-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 conflicts with the language definition, but may allow greater optimization to occur, particularly vectorization. This option may not be used if the formal parameter itself is assigned by the function (e.g., `formalParameter = &x[10]`); assignments may be made to the elements of the formal parameter (e.g., `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 great optimization to occur, particularly vectorization. This option may not be used if the formal parameter itself is assigned by the function (e.g., `formalParameter = &x[10]`); assignments may be made to the elements of the formal parameter (e.g., `formalParameter[10] = x[10]`).

`-alias cautious` Performs most optimal aliasing algorithm. 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 may only reference objects of that same type. For example, a pointer to an object of type `float` may not 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 may not function correctly.

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

`-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 effective only at levels `-O2` and `-O3`.

`-ep` *n* Optimizes the code for *n* processors, but will 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 the `-O3` option is also specified. 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 can be overridden by the `-O` options.

`-O` The highest scalar optimization level available, `-O1`.

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

If this option is not specified, the compiler performs no machine-independent optimization. The -O3 option is available only on the C2 computers, although it may degrade performance if the program is executed on a C210 computer.

-rl Performs loop replication optimizations (loop unrolling, dynamic code selection) on loops selected by the compiler on the basis of profitability. This option is equivalent to -ds -ur. This option is only effective at optimization levels -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 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 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.

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

The executable program that is created by this command line takes advantage of the instruction set and the availability of four processors of the C240 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/convex/oldvc is used. For example, the command **cc -B/usr/new** invokes `/usr/new/cocc` and `/usr/new/cpp` instead of the default `/usr/convex/oldvc/cocc` and `/usr/convex/oldvc/cpp`.

`-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 module 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, several additional compiler options are recognized by CONVEX C. 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:

Symbol	Description
_CONVEX_SOURCE _POSIX_SOURCE	These symbols are predefined in extended mode. In the conforming mode (`-std`) you 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 floating-point mode; _IEEE_FLOAT_ is defined in IEEE mode. See the description of the `-fi` and `-fn` compiler options for more information.
__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. This definition may not be removed by the `-U` option or `#undef`.
__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 the backward-compatible mode (`-pcc`). Its use is obsolescent; the symbol __convexc__ should be used instead.
__convex__ __unix__	These symbols are defined in the ANSI C conforming modes.
convex unix	These symbols are defined in the backward-compatible mode.
__LINE__	The line number of the current source line. This is predefined in all modes. This macro can be modified with the `#line` preprocessor directive in the ANSI C modes.
__FILE__	The name of the source file being compiled. This is predefined in all modes. This variable can be modified with the `#line` preprocessor directive in the ANSI C modes.
__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.
__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.

Other names beginning with "__" may be predefined by the C compiler. 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).

Linker Usage

The `cc` command line invokes the linker directly. Several compiler options translate into linker options. It is also possible to include additional linker options on the `cc` command line. CONVEX recommends always using the appropriate compiler to invoke the linker 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 linker. The compiler flags `-fi` and `-fn` are passed to the linker (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 linker 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 linker.

The following options, when present on the `cc` command line are passed to `ld` (see 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. Linker options which require a value (e.g., -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 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. For example, `-link -v3.2.8.5 -link -link` passes the flag `-v3.2.8.5` to the linker 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.

Do not use `-lc` and `-pcc` on the same `cc` command line; they are not compatible with each other. Also, it is not necessary 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 takes highest priority and CCOPTIONS take the 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 displayed by the compiler 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 the *ConvexOS Programmer's Reference* (see `csh`(1) and `sh`(1)). 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`(1) utility described in Appendix B, "Reporting Problems."

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 standard error. 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 earlier. 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 file.o
```

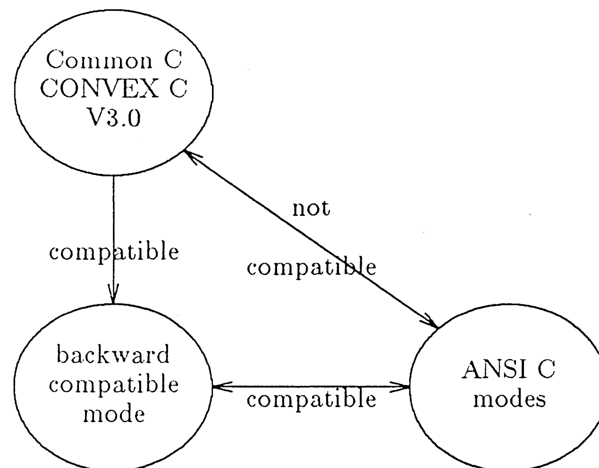
The first command line requires the file to be ANSI C compatible and to include POSIX header file information, the second command links the resulting object file with the default libraries.

For more information on the subject of mixed compatibility modes, refer to the *CONVEX C Language Reference Manual.*

Object File Compatibility

Object files and libraries created with CONVEX C V3.0 and the Common C compiler may continue to be used with CONVEX C V4.0. 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, object files created by the Common C compiler can be linked into applications that are compiled in the backward-compatible mode of CONVEX C. But no object files created in either the ANSI C modes or the backward-compatible mode of CONVEX C V4.0 may be linked into applications compiled with either the Common C compiler or CONVEX C V3.0.

Notes and Cautions

Scattered through the documentation are various notes and cautions on different subjects. They are referenced in each document index as well as in the master index.

Chapter 3
Program Development Tools

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

- `lint`: checks for errors not detected by the compiler
- `make`: eases program compilation 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 may cause bugs or reduce the portability of a program. It also checks the type usage of a program more strictly than C compilers do. Some errors that are currently found include unreachable statements, loops not entered at the top, and automatic variables declared and not used. Moreover, the usage of functions is 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 type checking of the compiler and `lint` is reduced. One improvement introduced by the ANSI C standard is increased type checking. But even though the compiler can perform type checking in one compilation unit, type checking across compilation units is not performed. 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("%d", 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 the file one.c, `x` is declared to be a `short int` whereas in file two.c, `x` is declared to be a `long int`.

Another feature of `lint` is that it can be used to reduce the amount of work that the compiler must do. 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 statement is recognized as unreachable. Dead code that `lint` discovers may be the result of a logical error in the program design.

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

`-D -U -I`	Same as for the `cc` compiler command.
`-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.
`-h`	Apply heuristic tests to attempt to find bugs, and improve style.
`-n`	Do not check compatibility to the standard library.
`-p`	Attempt to check portability to the IBM, VAX, and GCOS dialects of C.
`-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 referred to, but never used, by external declarations.

`lint` also includes some directives that can be included in the comments of a source file. These directives change the behavior of the `lint` utility. They include:

/*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.
/*NOSTRICT*/	Shuts off strict type checking in the next expression.
/*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 this file.

For information on this utility, refer to the man pages and *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 12 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:

```
#
# Make file for a short example.
#
myprog: myprog.o second.o
        cc myprog.o second.o -o myprog

myprog.o: myprog.c local.h
        cc -c myprog.c

second.o: second.c local.h pivot.h
        cc -c second.c
```

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. The seventh line 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 eighth line 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 the 11th and 5th lines 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.

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 can make programs easier to read; it converts different styles into one style.

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 virtually unreadable; it has no white space to help identify the hierarchy of the program. However, the following command line helps to untangle this program.

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

This command line processes the input file, indentin.c, according to the options in the remainder of the line, and produces the output file indentout.c. The `-bad` option specifies that there should a blank line follows every block of declarations; the `-bc` option specifies that a new line follows each comma in a declaration, and the `-di8` option specifies that 8 spaces separate a declaration keyword and the variable that it declares.

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

The `indent` utility provides many more options that can be used to format a program. For example, the `-troff` option formats the input file for the `troff`(1) utility. Consult `indent`(1) for more information on this utility.

error Utility

`error` analyzes and optionally inserts the diagnostic error messages produced by compilers and language processors, 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 error messages and source code to be viewed 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 language processor that produced each error message.
- Source file and line number to which the error message refers.
- Whether to ignore the error message.

`error` then inserts the (possibly slightly modified) error message 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. `error` touches source files only after all input has been read. Several options may be used with this utility:

- `-q` Confirm any potentially dangerous action (such as touching a file) or verbose action.
- `-t` *suffixlist* Do not touch files whose suffixes appear in the suffix list. The suffix list is dot-separated, and "*" wildcards are accepted.
- `-v` After all the files have had the errors inserted, initiate the `vi`(1) editor such that the first file with an error in it is being edited.
- `-T` Produce terse output.

For example, the following `csh` command line checks the syntax of a C source file, inserts any error messages into the source file, and places the source file in the `vi`(1) editor if errors are detected.

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

The equivalent command for the Bourne shell (`sh`(1)) is:

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

Chapter 4
Debugging Programs

Introduction

This chapter introduces utility programs that can be used 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:

- `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; detail can be found 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.

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:

- 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`(1)) 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:

- 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. These commands can be used 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 post-mortem dump (pmd(1)) 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 4-1. The short-form dump is the default.

Table 4-1: Postmortem Dump Contents

Type of Dump	Contents
Short Form	The signal that caused the program to abort. A runtime stack backtrace. The approximate source line location at which the exception occurred.
Long Form	The signal that caused the program to abort. A runtime stack backtrace. The approximate source line location at which the exception occurred. The contents of the machine registers. A dump of active local variables in each routine on the runtime stack. A dump of global, or common, variables. The region of disassembled object code where the exception took place. A 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 on this utility, refer to the *CONVEX Consultant User's Guide*.

Cross-Reference Generator

The cross-reference generator (`cref(1)`) 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 can be controlled by options specified on the `cref` command line.

Consider the following C source program, crefex.c.

```
#define loop_incr 200
#define array_size 1000

extern void sub1(int);
float a[array_size];

struct some_struct {
    int a;
    int b;
};

struct some_struct var_struct = {
    10,
    20
};

void main()
{
    int i;

    for( i=1; i<=array_size; i+=loop_incr )
        sub1(a[i]);
    var_struct.a = loop_incr;
    return(0);
}
```

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

cref crefex.c

is executed:

```
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.

There are several interesting things to note about `cref` output in the previous example. First, the object `some_struct` is defined on two lines. The `struct` is defined on line 7, but creates a variable on line 12. A peculiarity of the `cref` utility is that it considers both uses of `struct some_struct` to be a definition of that `struct`. 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, and the other is a member of a structure. Consequently, it is advisable to refrain from using several objects with the same name to avoid confusion. `cref` is useful as long as its limitations are kept in mind.

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`(1)) 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 the values of all variables. Using `adb`, you can:

- 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.

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 on its use, refer to the *CONVEX adb (Assembly-Language Debugger) User's Guide.*

Chapter 5
Profiling Programs

Introduction

This chapter discusses several tools that can be used 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 different profilers that allow you to monitor the performance of your program. These profilers are:

- 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 an option described in Table 5-1.

Table 5-1: 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 then 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 then be processed by the `bprof` profiler to display the source-level execution counts.
`-pg`	Produces instrumentation in the manner of `-p`, 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 on these optional profilers, examine the *CONVEX Consultant* package.

CXpa Profiler

The CONVEX C compiler supports the CXpa profiler (Version 2.0) on the CONVEX C200 series of computers. This is an optional product. CXpa V1.0 does not support C.

CXpa takes advantage of CONVEX hardware, providing a more accurate profiler than `prof` and `bprof`. This performance analyzer is described in detail in the *CONVEX Performance Analyzer (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:
  - Total times the routine is called.
  - Total wall-clock time spent in the routine and percentage of program total wall-clock time.
  - Total CPU time spent in the routine and percentage of program total CPU time.
  - Net CPU time spent in the routine and percentage of program total 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 can be used to determine 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 6
Optimizing C Programs

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

There are two versions of the CONVEX C V4.0 compiler: a scalar optimizing version bundled with each CONVEX machine and a vectorizing/parallelizing version, which is an optional product. Vector and parallel optimizations cannot be performed by the scalar compiler; compiler options associated with vectorization and parallelization are ignored by the scalar compiler.

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

Introduction

Optimization is the process of transforming functional code into an executable program that returns correct answers in less time. The CONVEX C V4.0 compiler automatically transforms your code to:

- Eliminate unnecessary operations.
- Perform operations in a more efficient order.
- Replace certain operations with faster ones.
- Take advantage of the CONVEX machine architecture.

These transformations operate at several different optimization levels. Each level adds a new degree of optimization to the 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.

Machine-dependent optimizations are performed at level `-O0` and higher, and take advantage of the CONVEX machine architecture by making efficient use of registers and functional units.

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 has only one entrance and one exit. The `-O0` command line option causes the compiler to perform basic-block scalar and machine-dependent optimization. Function-level scalar optimizations involve transformations of code within a single program unit (i.e., main program or function). 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. In a loop that, for example, 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, scalar, and machine-dependent optimization.

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, scalar, and machine-dependent optimizations are performed.

Scalar Optimization

The CONVEX C V4.0 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. A basic block is 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 attempts 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 CONVEX compiler attempts 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 may 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 may transform integer multiplication to 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. Following is a description of the various basic-block optimizations that can be performed:

- 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—The compiler recognizes a common subexpression and retains its value in a register to avoid repetitious calculations.
- 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 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 attempts to replace time-consuming operations with those that execute faster.

Function-Level Optimization

Function-level optimization is machine-independent scalar optimization that is 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 on the program being compiled. Following is a description of the various function-level optimizations that can be performed.

- 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, the statement `X = Y;` may allow the compiler to 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 V4.0 compiler automatically optimizes a program to take advantage of the CONVEX C Series architecture. 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 in addition to scalar optimizations.

The compiler reorders the statements and instructions of a program to make it easier to vectorize. The following explains the most important of these transformations:

- 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 of which has an iteration count that never exceeds 128 and 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 proper 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 right-most 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—Loop induction variables that are not incremented on every iteration are called conditional induction variables. 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 fully 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` directive.
- Loops that contain `goto` statements.
- Loops that contain real recurrences.

Parallel Optimization

The CONVEX C V4.0 compiler parallelizes and vectorization 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 turnaround 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 may 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` directive. This directive forces the compiler to parallelize the immediately following loop, regardless of potential dependencies the compiler may detect. Certain actual dependencies (such as from one scalar to another) cause the compiler to ignore the directive. Also, functions called from within a parallelized loop should be compiled using the `-re` option. Compiled in this way, each invocation of the function 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 directives, refer to the section in this chapter entitled "User-Directed Optimization."

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 may 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` directive. This directive 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` directive and other optimization directives, refer to the section entitled "User-Directed Optimization" in this chapter. Refer to the *CONVEX C Optimization Guide* for more information on loop-carried dependencies.

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 14) rather than being 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 all -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
   8-2    k       Scalar

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

The line numbers in the `Line Num.` column are actually number pairs, indicating that at least one loop was distributed. In this example, loop distribution results in the creation of 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 the following transformations were performed:

- The i loop at line 10 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 distributend 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 directives 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 directives. Table 6-1 summarizes the optimization options. If you do not specify one of the optimization options on the command line, the compiler defaults to the -no level.

Table 6-1: Optimization Options

Option	Explanation
-O3	Perform parallelization, vectorization, non-local scalar optimization, local scalar optimization, and machine-dependent optimization.
-O2	Perform vectorization, non-local scalar optimization, local scalar optimization, and machine-dependent optimization.
-O1	Perform non-local scalar optimization, local scalar optimization, and machine-dependent optimization.
-O0	Perform local scalar optimization and machine-dependent optimization.
-no	Perform minimal machine dependent optimization.

Optimization Directives

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

/***$dir** *directive* [, *directive*]*/

The directive must begin with the characters /***$dir**, followed by one or more of the directives summarized in Table 6-2, 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.

The scope of a compiler directive is the source file in which it appears. A directive must appear within a function body; a directive must be followed by a statement, even if it is the null statement (;). Because the compiler ignores comments, you may surround directive lines by any number of comment lines. Do not, however, include other comments within the same comment delimiters as a directive; use a separate set of /* */ delimiters.

Table 6-2: Optimization Directives

Directive	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_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	Disregard 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, if possible. This is useful with nested loops.
prefer_vector	Identifies to the compiler the loop you prefer to have vectorized, if possible. 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.

The scope of a loop optimization directive is the loop immediately following the directive; the scope does not, however, apply to loops nested therein.

Certain combinations of directives are invalid when used on the same loop and will cause the program unit to be rejected by the compiler. Table 6-3 lists the invalid combinations.

Table 6-3: Restrictions on Directive Use

Directive	Cannot be used with
`force_parallel`	`scalar, select, synch_parallel, unroll`
`force_vector`	`pstrip, scalar, select, synch_parallel, unroll, vstrip`
`pstrip`	`force_vector, scalar, unroll`
`scalar`	`force_parallel, force_vector, pstrip, select, synch_parallel, vstrip`
`select`	`force_parallel, force_vector, scalar, unroll`
`synch_parallel`	`force_parallel, force_vector, scalar, unroll`
`unroll`	`force_parallel, force_vector, pstrip, select, synch_parallel, vstrip`
`vstrip`	`force_vector, scalar, unroll`

The directives are explained in greater detail in the *CONVEX C Language Reference Manual.*

APPENDIX A
Error Messages

This appendix describes error messages encountered when a C source file is compiled. 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 some error messages specific to each compatibility-mode. For example, the backward-compatible mode does not detect ANSI C syntax errors. Choose the compatibility mode required for your application.

Compiler Diagnostic Options

Many compiler options described in Chapter 2, "Compiler Fundamentals," control the diagnostic output of the compiler. For example, `-w` suppresses all warning messages.

A powerful diagnostic option is `-d` because this 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.

Three ways to use this option are:

- To remove a message.
- To convert a message to a warning message.
- To convert a message to an error message.

Error messages prevent the creation of an executable program; warning messages do not.

For example, to permit storage class specifiers to follow type specifiers without generating a message, compile with:

```
-d obsolescent
```

or to generate a warning message, use:

```
-d obsolescent=w
```

Note	Converting an error message to a warning message or removing it entirely may cause undefined behavior in the compiler. Error messages exist for a purpose and should be manipulated infrequently.

The messages that can be manipulated are listed below. They are accompanied by a description of the condition which 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 may be the default. **Removing a diagnostic message does not impact the behavior of the compiler; it merely prevents the message from being displayed.**

`arg_ptr_qual`	Detect actual function parameters that do not have the same type qualifiers as those declared in the function prototype.
`arg_ptr_ref`	Detect actual function parameters that are not the same pointer type as those declared in the function prototype.
`class_ignored`	Use an explicit storage class to declare a tag or `enum` member.
`const_not_init`	Detect uninitialized constant variables.
`division_by_zero`	Detect division by zero.
`dollar_names`	Use the dollar sign symbol, `$`, in identifiers.
`escape_range_sequence`	Use an escape sequence integer constant greater than 0xff.
`float_suffix`	Use floating-point suffixes, f, F, l, or L.
`function_storage_class`	Declare an `auto`, `static`, or `register` function.
`func_ptr_math`	Use function pointer arithmetic.
`implicit_decl`	Use implicit function declarations.
`incomplete_enum`	Declaring incomplete `enum` tags such as `enum x;` before defining `x`'s members.
`integer_overflow`	Use an integer constant larger than 64 bits.
`long_long_suffix`	Use the integer literal suffixes LL or `ll`.
`nothing_declared`	Use empty declarations such as `int;`.
`null_effect_expression`	Eliminate intrinsic math function calls that have no effect.
`obsolescent`	Use obsolescent features such as embedding storage class specifers in type specifiers.
`old_form_assign`	Use old forms of assignment operators such as =+ and + =. Ambiguous assignments may result. For example, =- could be = followed by the unary minus sign or the -= operator.
`old_octal_digits`	Use the digits 8 and 9 in octal constants.
`sizeof_bitfield`	Use bitfield operands in the sizeof operator.
`strict_syntax`	Place a semicolon after the last member in a `struct` or `union` declaration or placing a comma after the last enumeration constant in an `enum` declaration list.
`unsigned_suffix`	Use the integer literal suffixes U or `u`.

The default settings of these diagnostic conditions in the four compatibility modes of CONVEX C are listed in Table A-1.

Table A-1: Diagnostic Condition Default Settings

Name	Compatibility Mode			
	Extended	Conforming	Strict	Backward-Compatible
arg_ptr_qual	warn	warn	warn	warn
arg_ptr_ref	warn	warn	warn	warn
class_ignored	warn	warn	warn	-
const_not_init	warn	warn	warn	warn
division_by_zero	warn	warn	warn	warn
dollar_names	-	warn	warn	-
escape_range_sequence	-	warn	warn	-
float_suffix	-	-	-	error
func_ptr_math	error	error	error	error
function_storage_class	error	error	error	-
implicit_decl	-	-	-	-
incomplete_enum	error	error	error	-
integer_overflow	-	warn	warn	-
long_long_suffix	-	warn	warn	-
no_external_declaration	-	warn	warn	-
nothing_declared	warn	warn	warn	-
null_effect_expression	warn	warn	warn	warn
obsolescent	-	-	warn	-
old_form_assign	error	error	error	warn
old_octal_digits	error	error	error	error
sizeof_bitfield	error	error	error	-
strict_syntax	warn	warn	warn	-
unsigned_suffix	-	-	-	error

The – character indicates that the default for the diagnostic condition is to generate no message.

Table A-2 is an index of the location for each of these diagnostic messages. The messages are discussed in greater detail in the following pages.

Table A-2: Index to Diagnostic Messages

Name	Page	Name	Page
arg_ptr_qual	A-44	integer_overflow	A-10
arg_ptr_ref	A-4	long_long_suffix	A-22
class_ignored	A-38	no_external_declaration	A-43
const_not_init	A-10	nothing_declared	A-26
division_by_zero	A-10	null_effect_expression	A-13
dollar_names	A-12	obsolescent	A-27
escape_range_sequence	A-13	old_form_assign	A-28
float_suffix	A-14	old_octal_digits	A-27
func_ptr_math	A-32	sizeof_bitfield	A-37
function_storage_class	A-15	strict_syntax	A-26
implicit_decl	A-21	unsigned_suffix	A-22
incomplete_enum	A-13		

Error Message Catalog

All the messages are listed in the following pages in alphabetical order. In determining the keyword used to sort the messages, the following words were ignored:

- `cc:`
- `a`
- `an`
- `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.

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. For 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`.

It is possible for the actual parameter or formal parameter to be a `void` pointer. A `void` pointer is a generic pointer that is compatible with any pointer type. For example:

```
/* compile with cc -d arg_ptr_ref=e file.c */
extern int func1(void *c);

main()
{
     int a;
     int *b;

     a = func1( b );
}
```

`cc: Argument to MAX_TRIPS directive must be greater than zero.`

The argument to the `max_trips` directive must be an integral constant greater than zero.

`cc: Argument to MAX_TRIPS` *argument* `is not an integer constant.`

The directive `max_trips` accepts only one integer constant as an argument. Only decimal, octal, or hexadecimal integral constants greater than zero are accepted.

`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 array is declared with an incomplete type. Incomplete types do not provide enough information to enable a compiler to determine their size. For example:

```
struct some_s array[10];
```

will generate this message if no `struct` has been defined with the tag `some_s`. This condition is not detected 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 that is returned by the functions.
array-name	Is the array name.
num	Is the number of elements in the array.

For example,

```
int *(*funks[5])();
```

declares an array of five pointers to functions that each return a pointer to an integer.

`cc: Attempt to modify an object with const-qualified type`

Identifiers that are qualified by the `const` qualifier can not be assigned a value, except when they are initialized. For example:

```
const int x;
int y;

x = y;
```

generates this message. One way to initialize a `const` qualified type is:

```
const int z = 5;
```

`cc: automatic aggregates may not be initialized.`

This message only occurs in the backward-compatible mode. An aggregate is a `struct` or a `union` data type. An aggregate is automatic if its storage class is automatic. 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 directive indicated in the error message. Please consult the *CONVEX C Language Reference Manual* for assistance with the directive.

`cc: begin_tasks directive with no end_tasks`

The optimization directive `begin_tasks` tells the compiler to generate parallel code for the immediately following series of tasks. The `next_task` directive 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` directive. These directives tell the compiler that certain nonloop sections of code can execute safely in parallel.

The following code fragment will cause the compiler to generate this error message at optimization level -O3:

```
/* $dir begin_tasks */
task1();
/* $dir next_task */
task2();
```

Correct this error by appending the `end_tasks` directive, as in:

```
/* $dir begin_tasks */
task1();
/* $dir next_task */
task2();
/* $dir end_tasks */
;
```

Remember that there must be a statement after the `end_tasks` directive, even if it is only the null statement (;).

`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` that is passed in a function call. For 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 as in:

```
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. For 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: 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 use -> or . on type whose members are not defined.`

This message occurs when the left operand of the -> or . operators is declared with an incomplete `struct` or `union`. A structure is incomplete if it has not had its 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 incomplete structure.

`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 detects an error, it makes an assumption of what the programmer intended before it continues compiling. If that assumption is incorrect, the compiler may detect errors that don't exist. This *cascade effect* may prevent the compiler from checking the entire source file. The removal of the initial error may remove later errors.

```
cc: can't take the address of a variable declared register
```

Variables that have the `register` storage class may be stored by the compiler in a machine register. Because a register has no address, the following code fragment is illegal:

```
register int var;
int * pvar;

pvar = &var;
```

The semantics of an identifier declared with the `register` storage class are different in the backward-compatible mode and the ANSI C modes. In the backward-compatible mode, the variable must be assigned to a register if one is available, however in the ANSI C modes, such an assignment is not required.

```
>>>>>   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 that is embedded in the C compiler. Please contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Consult contact(1) or the *Reporting Problems* appendix in any CONVEX manual to learn how to contact TAC.

```
cc: case statement not in switch statement.
```

The `case` statement is limited to use in `switch` statements. This message often indicates misplaced braces.

```
cc: character constant is too long. Max length is 8.
```

The maximum length of a character constant is 8 characters. For example,

```
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.

```
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. For example:

```
/* compile with cc -d division_by_zero=e file.c */
int x = 3/0;
```

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.
For example:

```
/* compile with cc -d integer_overflow=e file.c */
char x = 0xffffffffffffffff1;
```

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. Please contact your system manager or the CONVEX Technical Assistance Center (TAC). Consult contact(1) or the *Reporting Problems* appendix in any CONVEX manual to learn how to contact 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. For example:

```
/* compile with cc -d const_not_init=w file.c */
main()
{
    const int x;
}
```

produces this message.

`cc: continue statement not in do, for, or while statement.`

`continue` statements may only occur within a `do`, `for`, `switch`, or `while` statement. For example,

```
main()
{
    continue;
    return(0);
}
```

is one example of this error. Correct this error by deleting the `continue` statement.

`cc: CPU time limit exceeded`

This message occurs when the compiler exceeds the maximum amount of cpu time that is 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: 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 may not be initialized when the declaration appears in a function definition. For example:

```
int main(){
    extern int var = 4;
}
```

`extern` variables may be initialized at file scope only in the ANSI C modes.

`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 non-portable use of the dollar sign character. For example:

```
/* compile with cc -d dollar_names=e file.c */
int heavy$;
```

produces this message.

`cc: duplicate case label.`

No duplicate `case` labels in a `switch` statement are permitted.

Some code that generates this message is:

```
#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 message.

`cc: duplicate definition: label already defined.`

The scope of a label is an entire source file. This message indicates that duplicate labels have been defined in one source file.

`cc: embedded begin_tasks directives disallowed`

The following code fragment is illegal:

```
/* $dir begin_tasks */
task1();

/* $dir next_task, begin_tasks */
    taskA();

    /* $dir next_task */
    taskB();

/* $dir end_tasks, next_task */
task2();

/* $dir end_tasks */
;
```

Embedded `begin_tasks` directives 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` directive must be preceded by a `begin_tasks` directive. The `end_tasks` directive 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`

Message Name: **incomplete_enum**

This is caused by using incomplete `enum` types such as `enum x;` before defining `x`'s members. For example:

```
/* compile with cc -d incomplete_enum=e file.c */
enum some_e x;
```

produces this message.

`cc: error 'message-name' can not be changed to warning message or suppressed.`

The indicated message can not be manipulated using the `-d` compiler option.

`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';
```

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. For example:

```
/* compile with cc -d null_effect_expression=e file.c */
#include <math.h>

main()
{
    sin(0.0);
}
```

`cc: extra actual parameters in call to function with prototype`

The number of parameters in a function call exceeded the number of parameters in the function prototype.

`cc: field size too large for field 'member name'.`

The number of bits in a bit field may not exceed the number of bits available in the data type that is used to declare the bit field. For 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: floating point literal value contains suffix`

Message Name: **float_suffix**

This diagnostic option detects usage 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. For example:

```
/* compile with cc -d float_suffix=w file.c */
float real = 4.5d;
```

produces this message. But:

```
/* compile with cc -d float_suffix=w -str file.c */
float real = 4.5d;
```

results in a syntax error.

`cc: formal parameter parameter number of function function name has no name`

All parameters in a function definition must have a name. For 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.

`cc: function as formal parameter illegal.`

Functions may not be parameters of a function. The following is illegal:

```
int funk(void);
int funk2( int funk( void ) );
```

Functions are passed as pointers. The correct code is:

```
int funk( void );
int funk2( int (*funk)(void) );
```

`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. For example:

```
int main(){
    struct unknown_s func(void);
    (void) func();
}
```

In this example, `unknown_s` is a `struct` with an unknown size.

However, while it is illegal to call a function that has an incomplete return type, it is not illegal to declare one.

`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`

Message Name: **function_storage_class**

ANSI C does not permit functions declared in block scope with the `auto`, `static`, or `register` storage classes. For example:

```
/* compile with cc -d function_storage_class=e file.c */
void func()
{
    auto some_f();
}
```

produces this message.

`cc: function` *function name* `may not be initialized`

Functions can not be initialized. However, it is possible to initialize a pointer to a function.

`cc: function prototypes are not supported in this environment`

Function prototypes are not supported in the compatibility mode in which the compiler was run. A function prototype is a function declaration in which the parameter types are declared, as in:

```
int func(float param1, int param2, char param3);
```

This prototype declares a function with parameters of type `float`, `int`, and `char`, respectively.

`cc: function returning array illegal.`

A function cannot return an array. For example,

```
int funk( void )[];
```

is an illegal declaration of a function that returns an array of integers. But functions can return the address of an array, as in:

```
int ( *funk( void ) )[];
```

which 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, as in:

```
int (*func(void))(void);
```

In these examples, function `func` and the function it returns have no parameters.

`cc: function 'function name' redeclared.`

This message occurs when a function declaration or use does not match its definition. For example:

```
int main()
{
    int num = f();
    return(0);
}

float f()
{
    return(1.0);
}
```

In this example, when function `f` is first encountered, it is implicitly declared as a function returning type `int`. But the function definition for `f` indicates that it returns type `float`. This discrepancy causes the message to occur.

`cc: greater than 255 tasking directives`

A maximum of 254 `next_task` directives may be used with each `begin_tasks` directive. Correct the error by dividing the directives into blocks of 255.

`cc: ignoring initializer on 'extern' declaration of 'identifier'.`

This message only occurs in the backward-compatible mode of the compiler. This feature is maintained to remain compatible with previous versions of the compiler which *ignored* the initialization of external variables. For example, in:

```
extern int i = 5;
```

the initialization of the external variable 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. Consult *CONVEX ANSI C Concepts* document to understand all the ramifications of this choice.
- Use the external storage class specifier but initialize the identifier in the body of a function.

`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 floating point constant suffix`

The following floating-point suffixes are available in all modes of the compiler:

- F or f, `float` suffixes.
- L or l, `long double` suffixes.

The D or d `double` suffixes are available only in the extended mode.

`cc: illegal initialization.`

Permitted initializations:

- Static variable with a constant expression.
- Automatic variables with an expression.
- Floating-point variables with an arithmetic 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 pointer subtraction.`

This message occurs when pointers of different types are subtracted from each other, as in:

```
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 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 in that are defined in other compilation units. The `extern` storage class is the default storage class for functions.

`cc: illegal storage class for parameter declaration.`

The only storage classes that are available for parameter declarations are `auto` and `register`.

`cc: illegal struct or union combination.`

Structures that are assigned to each other must be declared with the same structure definition. An example that produces the error is:

```
struct {int *r;} *p;
struct {int *r;} *q;

p = q;
```

If the identifiers `p` and `q` are declared with the same definition, as in:

```
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 may 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. However, other 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, as in:

```
typedef int f();
volatile f x;
```

Correct this error removing the offending 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. For 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, as in:

```
const int * const 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, as in:

```
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. For 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, as in:

```
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, as in:

```
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. For 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. For 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`. For 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`. For 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. For example:

```
struct {
    int bits : 4;
} x = {{{3}}};
```

Correct this by removing the extra braces.

`cc: implicit declaration "extern int` *function_name*`()" supplied`

Message Name: **implicit_decl**

This message indicates that an implicit function declaration is detected. For example:

```
/* compile with cc -d implicit_decl=e file.c */
main()
{
    float rc = some_func();
}
```

produces this error message.

`cc: integer constant expected.`

Integer constants are required in some C contexts. One such context is the length of a bit field. For example:

```
int y = 3;
struct { int x : y; } z;
```

is illegal. The correct code is:

```
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 the use of the `long long` integer suffix (ll or LL). For example:

```
/* compile with cc -std -d long_long_suffix=e file.c */
long long int x = 4LL;
```

produces this message.

`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 suffix.`

The following integer 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: invalid wide character constant. 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. For example:

```
char wch = L'ss';
```

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 directive. The directives are described in the *CONVEX C Language Reference Manual.*

`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.

`cc: loop directive not textually immediately before loop.`

Directives that affect loops must immediately precede a loop. An example of a directive that causes this message is:

```
typedef int Mat[100][28];
int main(){
     int i,j,n,m=100;
     Mat a,b;

     /* $dir 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 directive 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 values.
- Pointer values.
- Enumerated values.
- Structure values.
- Union values.

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 will be generated. In the following 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.

Another example of a left operand of an assignment operator that cannot be modified is:

```
int funk(void), r(void);

main()
{
    funk = r;
    return(0);
}
```

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 will run 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.

`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. There are currently 8000 entries in the table.

`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. For example, the following command line:

```
cc -O2 file.c
```

will generate this 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: more optimization is possible if this function call has no side effects.`

Function calls that modify global variables limit the amount of optimization that can be performed. If the function indicated by this message has no side effects, the compiler can perform more optimization: declare the function with the `no_side_effects` directive. Consult *CONVEX C Language Reference Manual* for more information on directives.

`cc: Multiple argument lists in one declarator`

This message occurs when a function definition has more than one parameter list. For example:

```
void func(a, b, c)(d, e)
    int a, b, c, d, e;
{}
```

Correct this error by deleting the incorrect parameter list.

`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 nonzero integer less than the number of bits required to represent the type of the field.

`cc: Newline in string or character constant (inserting close quote)`

Newline characters cannot be embedded in string constants or character constants. The following example generates this message:

```
char c[] = "asdf
;lkj";
```

The compiler tries to recover from the error by inserting a close 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` directive must be preceded by a `begin_tasks` directive. The `next_task` directive 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. Please contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Consult contact(1) or the *Reporting Problems* appendix in any CONVEX manual to learn how to contact TAC.

`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. For example:

```
float value = 75.E;
```

is illegal.

`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. For example:

```
int value = 0x;
```

is illegal.

`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 include in the declaration, as in:

```
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 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. For 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. It is recommended that the dimension of the array be increased to allow this NULL character because it is required 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, as in:

```
int a[5][];
```

All indexes must be included in a declaration so that the compiler can allocate sufficient space for the array.

`cc: obsolescent feature at or before` *character*

Message Name: **obsolescent**

When this option is set, the compiler detects code that is considered obsolescent; such code may not be allowed in a future ANSI C standard. One code practice detected is the placement of storage-class specifier other than at the beginning of the declaration specifiers. For example:

```
/* compile with cc -d obsolescent=w file.c */
int extern x;
```

`cc: Octal constant contains digit 8 or 9`

Message Name: **old_octal_digits**

Old versions of C permitted the digits 8 and 9 to be used as octal digits. ANSI C does not permit this. The following example can be compiled in an ANSI C compatibility mode using the indicated command line:

```
/* compile with cc -d old_octal_digits file.c */
#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`. Similarly, the digit `8` is an octal `10`.

`cc: old form assignment operator used.`

Message Name: **old_form_assign**

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;
     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: 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 that is 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 will result in this message.

`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;
     /* $dir no_side_effects (bad) */
     z = func();
}
```

Correct this error by using the correct function name as an argument of the `no_side_effects` directive.

`cc: operands of` *operand* `have incompatible types.`

Each operator in C has requirements on what may be used as its operands. For example, operands of the − operator may both be arithmetic, may both be of the same pointer type, or the left operand may be a pointer only if the right pointer is an integer. The combination of any other data types will result 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 data type pointers that are compared are incompatible. For 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.
- Enumeration 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. The following example generates this error message:

```
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 is done to allow 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 you are advised to rewrite the code using the more traditional style 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 will correctly compile this code at each optimization level.

`cc: '`*-string*`' option not recognized`

This message occurs when the compiler does not recognize a command line option that is 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. For example:

```
cc -or function
```

The `-or` command line option does not have a `function` argument.

`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 used in arithmetic

Message Name: **func_ptr_math**

It is illegal to perform arithmetic on function pointers. This diagnostic option detects the use of function pointers in arithmetic. The following code:

```
/* compile with cc -d func_ptr_math file.c */
int noid(void){
    return(3);
}

int (*func)(void);
main()
{
    func = noid;
    func += 2;
    (*func)();
}
```

can produce an executable program which may or may not run.

```
>>>>>   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 characters for symbol table.
```

The storage for the compiler symbol table was exceeded. This may be caused by having too many characters in the source file. One solution is to split the source file into several smaller files. Please contact your system manager or the CONVEX Technical Assistance Center (TAC). Consult contact(1) or the *Reporting Problems* appendix in any CONVEX manual to learn how to contact 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 may be caused by having too many strings in the source file. One solution is to split the source file into several smaller files. Please contact your system manager or the CONVEX Technical Assistance Center (TAC). Consult contact(1) or the *Reporting Problems* appendix in any CONVEX manual to learn how to contact 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. For example:

```
int func(int a, int a, int a);
```

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. For example:

```
int func()
{
    return(2);
}

int func()
{
    return(2);
}
```

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. For example:

```
extern int i;

void f()
{
  i = 1;
}
static int i = 5;
```

This code can allocate memory for one or two variables called `i`. 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 as some type may not be redeclared as a `typedef` name. For 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:

```
int f(int ch);
```

either of these 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. For 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, then the use of pointers is suggested. For 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:` *'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 is 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.

`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. For example:

```
extern int funk();
static int funk();
main()
{}
```

or:

```
extern int x = 3;  /* initialization required for this error */
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; there can be no duplicate tag names for these type specifiers in the same scope.

For example:

```
enum bool {false, true};

main(){
struct bool { int value;};
enum bool;
}
```

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. The redeclaration must be of the same type. For 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: 'variable' redeclared: saw 'static' definition and then external definition.`

This message occurs when a `static` function or variable is redeclared with no storage class keyword. For 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: was previously declared as constant of enumerated type.`

An identifier declared as a constant of an enumerated type may not be redeclared in the same scope as another type. An example that generates this error is:

```
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 may not 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` directive may have the same numerical value. For example:

```
/*$dir 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. Such an action is impossible. The parameters of the `select` directive may never have the same value.

`cc: signed long long bitfields are not supported`

This message indicates that a `signed long long int` has been defined. This type of bit field is not supported.

`cc: sizeof(bitfield) disallowed`

Message Name: **sizeof_bitfield**

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, but this feature can be enabled with the `-d` compiler option. If so, then the sizeof operator returns the size of the object that contains the bit field. The following example displays the number of bytes required by an `unsigned long long int` bit field:

```
/* compile with cc -d sizeof_bitfield file.c */
#include <stdio.h>

main()
{
     struct {
          unsigned long long int bits : 64;
     } x;
     x.bits = 0x1;

     printf("sizeof bit field = %d\n", sizeof(x.bits) );
}
```

`cc: sizeof(function) disallowed.`

The operand of the sizeof operator may not be a function, but it may be a pointer to a function. For 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. The following is illegal:

```
int k = sizeof(void);
```

Correct this error by replacing `void` with the correct data type.

`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. While it is not illegal, the scope of the declaration does not extend to any variables declared with that `struct` type. For 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 not allowed in struct or union body`

The five storage classes:

- `auto`
- `extern`
- `register`
- `static`
- `typedef`

may not be used to modify a member declaration in the body of a `struct` or `union`.

`cc: string constant is too long. Max length is` *length*.

A string constant is an identifier that is initialized with a sequence of characters enclosed in double quotes. The identifier is a pointer to type `char` or an array of type `char`.

This message indicates that the sequence of characters is too long. If a variable must contain a string longer than the limit of a string constant, then use the strcat function to concatenate two or more string constants together when the program is executing. For example, if a string required is three times the maximum length of a string constant, then:

```
char *string  = "first third ...";
char *string2 = "second third ...";
char *string3 = "final third ...";

(void) strcat(string, string2);
(void) strcat(string, string3);
```

`cc: struct/union or struct/union pointer required.`

This message occurs when the . operator is used instead of the -> operator. One common way of getting this message is the following:

```
struct { int x; } *p;
int i;

i = p.x;
```

Because p is a pointer to a `struct`, the -> operator should be used instead of . in this example.

`cc: A structure or union member may not have a function type.`

The member of a structure or union may not be a function, but it can be a pointer to a function.

```
struct sample_s {
    int funk( void );
};
```

will generate the message, but

```
struct sample_s {
    int (*funk)( void );
};
```

will 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 may not 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.

`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;
```

will not generate any message.

`cc: a switch case label must be integer.`

A **case** label of a **switch** statement must be an integral type. The following example will generate this error:

```
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. The following example will generate this error:

```
float expn = 3.0;

switch( expn )
     :
```

`cc: Syntax Error at "`*string*`"`

There is a syntax error in the indicated file at *string*.

`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 directive. Please consult the *CONVEX C Language Reference Manual* for assistance with *directive*.

`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. For example:

```
int f(){
    enum some_tag { true, false };
    union some_tag;
}
```

To avoid this error, rename *tag* with a tag name that is 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. For example:

```
int f(){
    enum some_tag { true, false };
    struct some_tag;
}
```

To avoid this error, rename the *tag* with a tag name that is 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 compilation unit. For example:

```
enum color { red, white };
struct color {
    int hue;
    int intensity;
};
```

is a code fragment that will generate 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. For example:

```
cc -tm e1 file.c
```

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. For 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: There must be one and only one argument to MAX_TRIPS directive.`

The `max_trips` directive accepts only one integral constant greater than zero as an argument.

```
>>>>>   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 will reduce the chance of this error occurring. Please contact your system manager or the CONVEX Technical Assistance Center (TAC) when this error occurs. Consult contact(1) or the *Reporting Problems* appendix in any CONVEX manual to learn how to contact TAC.

`cc: too few actual parameters in function call`

This message indicates that the number of parameters in a function call did not agree with the number of parameters declared in its function prototype. For example:

```
void func(int a, int b);

main()
{
     func(3);
}
```

If the discrepancy is intentional, declare only the return type of the function as in:

```
void func();

main()
{
    func(3);
}
```

or, 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 detects an error, it makes an assumption of what the programmer intended before it continues compiling. If that assumption is incorrect, the compiler may detect errors that don't exist. This *cascade effect* may prevent the compiler from checking the entire source file. The removal of the initial error may remove later errors.

`cc: too many initializers.`

This message occurs when an array, `struct`, or `union` is initialized with too many values. For example, in the declaration

```
char arry[5] = "123456";
```

the array `arry` has five elements, but the string has 6 characters in it.
See also: NULL `at end of string initializer '`*characters*`' truncated.`

`Translation unit contains no external declarations`

Message Name: **no_external_declarations**

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 define a function type using `typedef`, as in:

```
typedef int func_t(int param);
```

But such a function type cannot be used to *define* a function. For 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. The above error example is fixed with:

```
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, specifically, its parameter list including the parentheses is missing. For example:

```
int func
{
}
```

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. For example:

```
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.`

Unlike other data types, types defined with *typedef* may not have an initial value. This message can be generated by:

```
typedef int var = 1;
```

The only way this error can be resolved is to initialize the variable of this type when it is declared or in the code itself. For 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.

These are objects which have no address, which is the function of the & operator.

`cc: Unexpected end of source file.`

The compiler encountered the end of the source file unexpectedly. Several causes of this error are

- End of source file in a comment.
- Mismatched pairs of { and }.
- Mismatched pairs of (and).

The last two problems may be caused by misplaced comments or conditionally compiled code.

`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, as in:

```
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, as in:

```
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.

`Unrecognized diagnostic name` '*identifier*' `-- ignored`

The *identifier* specified with the `-d` command line option was not recognized by the compiler. The recognized diagnostic names are defined at the beginning of this appendix.

`cc: Unrecognized Escape Sequence` '*sequence*'

There are three types of escape sequences 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. Consult *CONVEX C Language Reference Manual* for more information on escape sequences.

`Using -lc in -pcc mode causes incorrect library usage`

The `-lc` switch causes files to be linked with a library that is supplied for use with the extended mode of the compiler. Using it in the backward-compatible mode may cause linkage errors. In most cases removing `-lc` from the command line will allow linking to proceed normally and produce the correct result; cc automatically searches the correct library sets, so using `-lc` is not necessary.

`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 program:

```
register int z;
main()
{
}
```

generates this message.

`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`. For 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 no declaration is visible. Either the identifier was 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. For example:

```
void func()
{
    x = 5;
}
```

To resolve this, define the variable in a location that is visible to its location of use.

`cc: void type not allowed in this declaration.`

The `void` type may not be used to declare variables. For example:

```
void var;
```

`void` is typically used as the return type of a function, the base type of a pointer, or to indicate that a function that has no parameters.

`cc: void type used in expression.`

Declaring the elements of array to be type `void`, or defining the parameter of a function to have type `void` is illegal. For example:

```
void a[5];  /* or */
int func(void param);
```

Type `void` is illegal in these cases because the compiler cannot determine how much memory to allocate for the identifiers.

`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 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;
```

`cc: zero length character constant.`

Character constants must contain at least one character in a character sequence. The following is illegal:

```
int ch = '';
```

APPENDIX B
Reporting Problems

This appendix introduces the CONVEX Technical Assistance Center (TAC) and `contact` utility. The `contact` utility is an online system for reporting problems to the TAC. To learn `contact` by using it, enter **contact** at the system prompt and then answer the questions as they appear on the screen. To find out more about using `contact`, read through this appendix. It describes prerequisites and tips for using `contact` and the step-by-step process `contact` takes you through.

Technical Assistance Center

The CONVEX Technical Assistance Center (TAC) is staffed by technical specialists who can address diverse questions and problems that arise in a supercomputing environment. If you have a hardware, software, or documentation question, contact the TAC. This group stands ready to solve such problems.

The contact Utility

The TAC recommends using the `contact` utility to report a hardware, software, or documentation problem. The `contact` utility is an interactive program that helps the TAC track reports and route them to the CONVEX personnel most qualified to fix a problem.

After you invoke `contact`, it prompts you for information about the problem. When you finish your report, `contact` electronically mails it to the TAC. You are notified within 48 hours that the TAC has received your report.

Prerequisites

To use `contact` requires

- a UNIX-to-UNIX Communication Protocol (UUCP) connection to the TAC
- full path name of the program or utility in question
- version number of the program or utility in question

UUCP Connection

Before using `contact`, check with your system administrator to be sure there is a UUCP connection to the TAC. A UUCP connection allows files to be copied from one UNIX-based system to another. The `uucp` (UNIX-to-UNIX copy) command relies on either a dial-up or hard-wired UUCP communication line.

Finding the Program Path Name

To determine the full path name of the program or utility in question, use the `which` command. The next screen illustrates using the `which` command to find the full path name of the loader (`ld`) utility.

```
>which ld
/bin/ld
>
```

In this example, the full path name of the loader is /bin/ld.

For more information on the `which` command, refer to the which(1) man page. You can also use the `info` online information system. Enter **info which** at the system prompt.

If you use the C shell (`csh`), you can also use the `whence` command to find the program path name. The `whence` command works like `which`, but faster.

Finding the Program Version Number

To determine the version number of the program or utility in question, use the `vers` command. The next screen illustrates using the `vers` command to find the version number of the loader (`ld`) utility. Enter **vers**, then the path name of the program or utility.

```
>vers /bin/ld
/bin/ld: 7.0
>
```

In this example, the loader version number is 7.0.

For more information on the `vers` command, refer to the vers(1) man page. You can also use the `info` online information system. To do so, enter **info vers** at the system prompt.

Tips on Using the contact Utility

The `contact` utility is interactive and easy to use. This section lists tips to help use it efficiently. In particular, this section tells how to

- use a .contact file
- abort a contact session
- resubmit an aborted report
- suspend a contact session
- move from one prompt to another
- use tilde-escape sequences in the `contact` utility

Using a .contact File

When you invoke `contact`, it prompts for information regarding the problem. The first prompt is for your name, title, phone number, and company name. You can, however, create a .contact file to skip this first prompt. Follow these steps:

1. Create a .contact file in your home directory.

2. Enter your name, job title, phone number, and company name, each on a new line.

When you invoke `contact`, it automatically includes the .contact file as input for the first prompt and proceeds to the next prompt.

Aborting the Report

To abort a contact report, either press the interrupt key (usually CTRL-C) or choose the abort option when prompted by the `contact` utility. Using CTRL-C to abort does not save the contents of the report. Using the abort option saves the contents of the report in a file named dead.report in your home directory.

Submitting the dead.report File

After you abort a contact session, the `contact` utility saves the report in a file named dead.report in your home directory. Using the `contact` command with the `-r` option automatically merges the contents of the dead.report file into the new contact session. Enter

contact -r

and `contact` finds the dead.report file in your home directory and merges it into the contact report. You can then edit the report. When you end the editing session, `contact` returns to the final prompt, which asks you to review, edit, submit, or abort the report.

Suspending a Report

Sometimes it is necessary to stop in the middle of a contact report and return to the shell (for instance, to suspend the contact session to find the program path name or version number). To suspend the contact session, press CTRL-Z. To return to the contact session, enter **fg**. Using CTRL-Z and the `fg` (foreground) command lets you toggle back and forth between the `contact` utility and the shell. You cannot, however, use CTRL-Z and `fg` to toggle back and forth if you are using the Bourne shell (`sh`).

Ending a Response

The `contact` utility prompts for information pertinent to your hardware, software, or documentation question. Some prompts require one-line responses; to move to the next prompt, press RETURN. Other prompts require more than a one-line response; to move to the next prompt, press CTRL-D.

Tilde-Escape Sequences

The `contact` utility treats input beginning with a tilde (~) as a special sequence. The character following the tilde is considered a request for a special function. The following tilde sequences are recognized by `contact`:

~e start the text editor (defined in the EDITOR environment variable)

~h display a list of available tilde-escape sequences

~p print the contact report to the terminal screen

~r *filename* read the contents of *filename* as a response to the current prompt. Some prompts require only a one-line response. This tilde-escape sequence works only for prompts that allow more than a one-line response.

~~ insert a single tilde as the first character in the line

Using the contact Utility

The `contact` utility prompts for the following information:

- your name, title, phone number, and corporate name
- name and version of the product
- one-line summary of the problem
- detailed description of the problem
- priority of the problem
- instructions on how to reproduce the problem
- comments about the problem
- comments about the documentation related to the problem
- files to include in the contact report

The following is a step-by-step discussion of these prompts.

Step 1a To invoke the `contact` utility, enter **contact** at the system prompt. The system responds with a welcome message and a series of questions regarding your hardware, software or documentation question. The next screen illustrates the `contact` command and the system response.

```
>contact
Welcome to contact version 0.11 ()

Enter your name, title, phone number, and corporate name (^D to terminate)
>
```

Step 1b If there is a .contact file in your home directory, `contact` skips the first prompt. The next screen illustrates the `contact` command and the system response when a .contact file is in your home directory.

```
>contact
Welcome to contact version 0.11 ()

Enter the name of the product involved
>
```

Step 2 The `contact` utility prompts for the version number of the product. If you do not know the version number, use CTRL-Z to suspend the session. Use the `which` (or `whence` if you use `csh`) and `vers` commands to find the version number of the product. Use the `fg` command to return to the session and enter the version number in the form X.X or X.X.X.X.

Step 3 The `contact` utility prompts for a one-line summary of the problem. This summary is the subject header in any further correspondence regarding the problem. Please make this summary as descriptive as possible in one line.

Step 4 The `contact` utility prompts for a detailed description of the problem. Please make this description as complete as possible. Include source code and a stack backtrace when possible. (Refer to the adb(1) or csd(1) man page for information on obtaining a stack backtrace.) The more information you provide, the quicker the TAC can isolate and solve the problem.

Step 5 The `contact` utility prompts for the priority of the problem. The next screen illustrates this prompt and priority levels from which to choose; you must enter a priority number.

```
Enter a problem priority, based on the following:
1) Critical    - work cannot proceed until the problem is resolved.
2) Serious     - work can proceed around the problem, with difficulty.
3) Necessary   - problem has to be fixed.
4) Annoying    - problem is bothersome.
5) Enhancement - requested enhancement.
6) Informative - for informational purposes only.
>
```

Step 6 The `contact` utility prompts for an explanation of how to reproduce the problem. Please include the command syntax and options you used and anything else you did to make the program run.

Step 7 The `contact` utility prompts for any other pertinent comments. Please include all relevant information.

Step 8 The contact utility prompts for suggestions regarding documentation supporting the product. Indicate whether documentation could be revised to address the problem.

Step 9 The contact utility asks for names of files necessary to reproduce the problem. The next screen illustrates the contact prompt and sample user response.

```
Are there any files that should be included in this report (yes | no)?
>yes
Please enter the names of the files, one to a line (^D to terminate)
>test.f
>~/subroutines/sub.f
>
```

> **Note** Tilde-escape sequences are not recognized in responses to this prompt. In contact, a tilde in this section means your home directory. This convention is based on use of the tilde for expanding file names in csh.

If files specified are small text files, they are automatically included in the contact report. If the files are too large to be included in this report, contact gives further instructions on how to submit these files.

To specify a directory, combine directory files into a single file using the tar command (refer to the tar(1) man page for further information) or enter each file name in the directory on a single line in the contact report.

Step 10 The contact utility prompts you to review, edit, submit, or abort the contact report. The next screen illustrates this prompt.

```
Please select one of the following options:
1) Review the problem report.
2) Edit the problem report.
3) Submit the problem report.
4) Abort the problem report.
>
```

Choose the number of the option you want to select. These options let you do the following:

Review review the text of the contact report. You are then prompted again to select an option.

Edit edit the text of the contact report. If you choose to edit the report, contact puts you in your default text editor.

Submit sends the report to the CONVEX TAC. You are notified within 48 hours that the TAC has received the report. This option exits the contact utility and returns you to the shell.

Abort saves the text of the report in a file named dead.report in your home directory. This option exits the `contact` utility and returns you to the shell.

Index

__convex__ macro ug-2-12
__convexc__ macro ug-2-12
__DATE__ macro ug-2-12
__FILE__ macro ug-2-12
__LINE__ macro ug-2-12
__STDC__ macro ug-2-12
__TIME__ macro ug-2-12
__unix__ macro ug-2-12
_CONVEX_FLOAT_ macro ug-2-12
_CONVEX_SOURCE macro ug-2-12
_convexvc macro ug-2-12
_IEEE_FLOAT_ macro ug-2-12
_POSIX_SOURCE macro ug-2-12
_setjmp function
    optimization with ug-A-30
() (function) operator, operand of ug-A-29
+ (unary plus) operator ug-A-28
- (unary minus) operator ug-A-28
-Enoposix, linker option ug-2-12
-Eposix, linker option ug-2-12
-p option ug-5-2
-pb option ug-5-2
-pg option ug-5-2
.contact file, skipping first prompt by using ug-B-3

A

adb debugger ug-4-4
aggregate
    definition ug-A-6
    initialization of automatic storage class ug-A-6
Alaska
    technical assistance for, how to obtain ug-viii
arg_ptr_qual
    compatibility mode actions ug-A-3
    error message ug-A-44
arg_ptr_ref
    compatibility mode actions ug-A-3
    error message ug-A-4
array
    character array initialization ug-A-27, ug-A-43
assembly-language debugger ug-4-4
associated documents
    ug-vii
    how to order ug-viii

B

basic block
    optimization ug-6-3
bibliography ug-vii
bit field
    data types ug-A-19
    initialization ug-A-21
    maximum size ug-A-14
    zero width ug-A-48

C

Canada
    technical assistance for, how to obtain ug-viii
CCLIBS environment variable ug-2-13
CCOPTIONS environment variable ug-2-13
character array
    initialization ug-A-27, ug-A-43
character constant
    initialization ug-A-9
    maximum length ug-A-9
    wide ug-A-22
    zero length ug-A-48
character set
    illegal characters ug-A-17
class_ignored
    compatibility mode actions ug-A-3
    error message ug-A-38
compatibility
    object file ug-2-15
compatible type
    description ug-A-30
    pointer ug-A-29
compiler
    definition ug-1-1
    mixed compatibility modes ug-2-15
compiler error
    example ug-2-14
    internal compiler error ug-A-10
    machine serial number mismatch ug-A-9
    no available memory ug-A-25
    parser stack exceeded ug-A-42
    program is too complex ug-A-32
compiler limits
    CPU time limit ug-A-11
    file size limit ug-A-14
    maximum number of scalars in function ug-A-24
compiling
    introduction ug-1-1
    multiple files ug-1-2
    one source file, example ug-1-2
    simple program ug-1-1
const_not_init
    compatibility mode actions ug-A-3
    error message ug-A-10
contact
    aborting the report ug-B-3, ug-B-7
    editing the report ug-B-6
    ending a response ug-B-4
    ending the report ug-B-6
    including files in the report ug-B-6
    invoking ug-B-1, ug-B-4
    prerequisites ug-B-1
    prompts ug-B-4
    reporting problems ug-B-1
    restrictions on tilde-escape sequences ug-B-6
    reviewing the report ug-B-6
    skipping first prompt by using .contact file ug-B-3

contact (cont)
step-by-step discussion of prompts ug-B-4
submitting dead.report file ug-B-3
submitting the report ug-B-6
suspending the report ug-B-3
tilde-escape sequences ug-B-4
tips on using ug-B-3
CONVEX consultant debugger ug-4-1
convex macro ug-2-12
CONVEX symbolic debugger ug-4-1
ConvexOS considerations ug-2-13
cref program ug-4-3
cross-reference generator ug-4-3
csd debugger ug-4-1
customer support
telephone number for ug-viii
CXpa profiler ug-5-2

D
dead.report file
submitting ug-B-3
using -r option to submit ug-B-3
debugger
adb ug-4-4
assembly-language ug-4-4
consultant package ug-4-1
csd ug-4-1
symbolic ug-4-1
tools ug-4-1
diagnostic messages ug-2-14
directives
lint utility ug-3-2
optimization ug-6-8
division_by_zero
compatibility mode actions ug-A-3
error message ug-A-10
documentation
ordering ug-viii
subscription service, how to apply ug-viii
dollar_names
compatibility mode actions ug-A-3
error message ug-A-12
dump
post-mortem ug-4-2

E
environment variables
CCLIBS ug-2-13
CCOPTIONS ug-2-13
environment, ConvexOS ug-2-13
error message
-d options ug-A-2
catalog ug-A-4
control ug-A-1
diagnostic options ug-A-1
error messages, controlled
arg_ptr_qual ug-A-3
arg_ptr_ref ug-A-3
class_ignored ug-A-3

error messages, controlled (cont)
const_not_init ug-A-3
division_by_zero ug-A-3
dollar_names ug-A-3
escape_range_sequence ug-A-3
float_suffix ug-A-3
func_ptr_math ug-A-3
function_storage_class ug-A-3
implicit_decl ug-A-3
incomplete_enum ug-A-3
integer_overflow ug-A-3
long_long_suffix ug-A-3
no_external_declaration ug-A-3
nothing_declared ug-A-3
null_effect_expression ug-A-3
obsolescent ug-A-3
old_form_assign ug-A-3
old_octal_digits ug-A-3
sizeof_bitfield ug-A-3
strict_syntax ug-A-3
error reporting ug-B-1
error utility ug-3-4
escape sequences
description ug-A-46
escape_range_sequence
compatibility mode actions ug-A-3
error message ug-A-13
example
array of pointers to functions ug-A-6
const-qualifier ug-A-44
function parameter is pointer to function ug-A-15
function returns pointer to array of int ug-A-16
function returns pointer to functions returning int ug-A-16
incomplete type ug-A-5
tasking directives ug-A-6
extern storage class
changing from extern to static ug-A-35
changing to static from extern ug-A-33, ug-A-34
initialization in function ug-A-11

F
figure
compiler and linker interactions ug-1-4
linking library routines ug-1-5
multiple source files ug-1-3
role of the compiler ug-1-1
sample program, file2.c ug-1-3
sample program, prog2.c ug-1-3
file name
naming conventions ug-2-2
flags
code generation ug-2-4
compatibility ug-2-6
diagnostic ug-2-6
miscellaneous ug-2-11
optimization ug-2-9

flags (cont)
  preprocessor ug-2-3
  utility support ug-2-7
float_suffix
  compatibility mode actions ug-A-3
  error message ug-A-14
func_ptr_math
  compatibility mode actions ug-A-3
  error message ug-A-32
function
  definition ug-1-1
  incomplete return type ug-A-15
  initialization ug-A-16
  parameter storage class ug-A-19
  static storage class ug-A-15
  storage class ug-A-18
function prototype
  definition ug-A-16
function_storage_class
  compatibility mode actions ug-A-3
  error message ug-A-15
function-level optimization ug-6-3
further reference ug-vii

H

Hawaii
  technical assistance for, how to obtain ug-viii

I

implicit_decl
  compatibility mode actions ug-A-3
  error message ug-A-21
incomplete type
  definition ug-A-5
  example ug-A-5, ug-A-45
incomplete_enum
  (See also: incomplete type) ug-A-13
  compatibility mode actions ug-A-3
  error message ug-A-13
`indent` utility ug-3-3
inhibitors of parallelization ug-6-5
inhibitors of vectorization ug-6-4
initialization
  character array ug-A-27
  character string ug-A-43
  function ug-A-16
  legal initializations ug-A-18
  `typedef` variables ug-A-45
initializers
  bit field ug-A-21
integer_overflow
  compatibility mode actions ug-A-3
  error message ug-A-10
integral constant expressions, defined ug-A-40

L

label, scope of ug-A-12
libraries, general ug-1-5
linker
  description ug-1-4
  specifying libraries ug-2-12
  usage ug-2-12
linker options
  `-Enoposix` ug-2-12
  `-Eposix` ug-2-12
  on `cc` command line ug-2-12
`lint` utility
  description ug-3-1
  directives ug-3-2
  options ug-3-2
long_long_suffix
  compatibility mode actions ug-A-3
  error message ug-A-22
longjmp function
  optimization with ug-A-30
loop-carried dependency ug-6-5
lvalue, definition of ug-A-23

M

machine-dependent optimizations ug-6-1, ug-6-2
`make` utility ug-3-3
messages
  compiler ug-2-14
  diagnostic ug-2-14

N

name space
  ordinary identifiers ug-A-36
  `struct`, `union`, and `enum` ug-A-35
  tags ug-A-41
no_external_declaration
  compatibility mode actions ug-A-3
  error message ug-A-43
note
  `-d` option ug-A-2
  converting error messages ug-A-2
  converting errors to warnings ug-A-2
  restrictions on tilde-escape sequences with `contact` ug-B-6
nothing_declared
  compatibility mode actions ug-A-3
  error message ug-A-26
null_effect_expression
  compatibility mode actions ug-A-3
  error message ug-A-13

O

object file compatibility ug-2-15
obsolescent
  compatibility mode actions ug-A-3
  error message ug-A-27

old_form_assign
compatibility mode actions ug-A-3
error message ug-A-28
old_octal_digits
compatibility mode actions ug-A-3
error message ug-A-27
optimization
basic-block ug-6-3
directives ug-6-8
function-level ug-6-3
machine-dependent ug-6-1, ug-6-2
options ug-6-7
parallel ug-6-2, ug-6-5
report ug-2-14, ug-6-6
scalar ug-6-1, ug-6-2
user-directed ug-6-7
vector ug-6-1, ug-6-4
options
code generation ug-2-4
compatibility ug-2-6
compatibility with other compilers ug-2-11
diagnostic ug-2-6
introduction ug-1-2
miscellaneous ug-2-11
optimization ug-2-9
preprocessor ug-2-3
utility support ug-2-7
options
-alias array_args ug-2-9
-alias cautious ug-2-9
-alias ptr_args ug-2-9
-alias standard ug-2-9
-alias worst ug-2-9
-asm ug-2-4
-B ug-2-11
-C ug-2-3
-c ug-2-4
-D ug-2-3
-d ug-2-6, ug-A-1
-db ug-2-8
-ds ug-2-9
-E ug-2-3
-ep ug-2-9
-ext ug-2-7
-extern distinct ug-2-4
-extern same ug-2-4
-fd ug-2-4
-fi ug-2-4
-float dp_const ug-2-4
-float dp_ops ug-2-4
-float sp_const ug-2-4
-float sp_ops ug-2-4
-fn ug-2-5
-I ug-2-3
-k ug-2-3
-no ug-2-9
-nw ug-2-6
-O ug-2-9
-o ug-2-11
-O1 ug-2-10
-O2 ug-2-10
options (cont)
-O3 ug-2-10
-O4 ug-2-10
-or ug-2-6
-p ug-2-8
-pa ug-2-8
-pab ug-2-8
-par ug-2-8
-parens explicit ug-2-5
-parens ignore ug-2-5
-parens implicit ug-2-5
-pb ug-2-8
-pcc ug-2-7
-pg ug-2-8
-re ug-2-5
-rl ug-2-10
-S ug-2-5
-sc ug-2-6
-std ug-2-7
-str ug-2-7
-string read_only ug-2-5
-string read_write ug-2-5
-tl ug-2-11
-tm ug-2-5
-U ug-2-3
-uo ug-2-10
-ur ug-2-10
-va ug-2-10
-vn ug-2-11
ordering documentation
how to ug-viii

P

parallel
optimization ug-6-2, ug-6-5
parallelization
description ug-6-2, ug-6-5
inhibitors ug-6-5
pmd utility ug-4-2
POSIX
definition ug-2-6
post-mortem dump ug-4-2
predefined symbols ug-2-12
profiler, CXpa ug-5-2
prototype, definition ug-A-16

R

Reader's Forum ug-viii
reentrant
use with -re ug-6-5
register storage class
legal uses ug-A-47
report, optimization ug-2-14
reporting problems ug-viii
runtime
messages ug-2-14

S

scalar optimization ug-6-1, ug-6-2
setjmp function
  optimization with ug-A-30
sigsetjmp function
  optimization with ug-A-30
sizeof operator
  bit field as argument ug-A-37
  function as argument ug-A-37
  `void` as argument ug-A-38
sizeof_bitfield
  compatibility mode actions ug-A-3
  error message ug-A-37
`static` storage class
  changing from `extern` to `static` ug-A-35
  changing to `static` from `extern` ug-A-33, ug-A-34
storage class
  function ug-A-18
  function parameter ug-A-19
  `register` ug-A-47
  `struct` or `union` ug-A-38
strict_syntax
  compatibility mode actions ug-A-3
  error message ug-A-26
string constant, definition ug-A-38
`struct`
  assignments ug-A-19
`switch case` label, data type ug-A-40
`switch` control expression, data type ug-A-40
symbolic debugger ug-4-1

T

table
  compiler options for profiling ug-5-2
  diagnostic condition default settings ug-A-3
  index to diagnostic messages ug-A-3
  optimization directives ug-6-9
  optimization options ug-6-7
  restrictions on optimization directive use ug-6-10
  type of post-mortem dump contents ug-4-2
TAC (Technical Assistance Center) ug-B-1, ug-viii
tasking directives
  example ug-A-7
  maximum number ug-A-17
technical assistance
  obtaining ug-viii
technical assistance center
  telephone number for ug-viii
technical assistance center (TAC) ug-B-1, ug-viii
tilde-escape sequences
  restrictions on use ug-B-6
  use in `contact` ug-B-4
trouble reports ug-B-1, ug-viii

U

unary minus
  operands ug-A-28
unary plus
  operands ug-A-28
`union`
  assignments ug-A-19
unix macro ug-2-12
UNIX-to-UNIX
  Communication Protocol ug-B-1
  copy command, `uucp` ug-B-1
unsigned_suffix
  compatibility mode actions ug-A-3
  error message ug-A-22
user-directed optimization ug-6-7
UUCP
  connection to TAC ug-B-1
`uucp`
  UNIX-to-UNIX copy command ug-B-1

V

vector
  optimization ug-6-1, ug-6-4
vectorization
  description ug-6-1, ug-6-4
  inhibitors ug-6-4
`vers` command
  using to find program version number ug-B-2
`void` data type
  typical uses ug-A-47

W

`whence` command
  using to find program path name ug-B-2
`which` command
  using to find program path name ug-B-2
wide character constant
  definition of ug-A-22

CONVEX C User's Guide
Document No. 720-000630-203, Fourth Edition

Reader's Forum

Please use this form to submit comments or questions concerning the clarity and service of this manual. Constructive critical comments are most welcome and help us continue in our efforts to generate quality customer documentation. Please list the page number for questions or comments.

____

____

____

____

____

____

____

____

From:

Name________________________ Title________________________

Company________________________ Date________________________

Address and Phone No.____

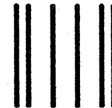
FOR ADDITIONAL INFORMATION OR DOCUMENTATION:

Location	Phone Number
From all locations in continental U.S.	1(800)952-0379
From locations in Alaska, Hawaii, & Canada	1(214)497-4379
From all other locations	Contact nearest CONVEX office

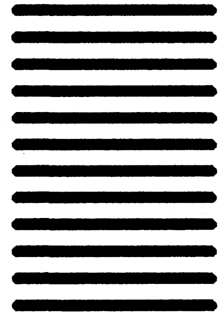
Direct mail orders to:

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851 USA

(Fold Here First)



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1046 RICHARDSON, TEXAS

POSTAGE WILL BE PAID BY ADDRESSEE

CONVEX Computer Corporation
Customer Service
PO Box 833851
Richardson TX 75083-3851
USA

(Fold Here Second)

(Tape or Staple)