Programmer's Guide

SPARCompiler™ C

Version 2.0

SunPro

A Sun Microsystems, Inc. Business

2550 Garcia Avenue, Mountain View, California 94043-1100

The OPEN LOOK and the Sun Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 (October 1988) and FAR 52.227-19 (June 1987).

The product described in this manual may be protected by one or more U.S. patents, foreign patents, and/or pending applications.

TRADEMARKS

The Sun logo, Sun Microsystems, Sun Workstation, NeWS, SunPro, and SunLink are registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Sun, Sun-2, Sun-3, Sun-4, Sun386*i*, SunCD, SunInstall, SunOS, SunView, NFS, and OpenWindows are trademarks of Sun Microsystems, Inc.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

PostScript is a registered trademark of Adobe Systems Incorporated. Adobe also owns copyrights related to the PostScript language and the PostScript interpreter. The trademark PostScript is used herein only to refer to material supplied by Adobe or to programs written in the PostScript language as defined by Adobe.

X Window System is a product of the Massachusetts Institute of Technology.

SPARC is a registered trademark of SPARC International, Inc. Products bearing the SPARC trademark are based on an architecture developed by Sun Microsystems, Inc. SPARCstation is a trademark of SPARC International, Inc., licensed exclusively to Sun Microsystems, Inc.

All other products or services mentioned in this document are identified by the trademarks, service marks, or product names as designated by the companies who market those products. Inquiries concerning such trademarks should be made directly to those companies.

Portions © AT&T 1983-1990 and reproduced with permission from AT&T.

Contents

Figures

Tables

Preface

The *SPARCompilers™ C 2.0 Programmer's Guide* is a reference guide to this implementation of the ANSI C language.

Operating Environment

The SPARCompiler C 2.0 compiler runs under two operating environments:

- SunOS™ 4.1.1 (and later) operating system
- SunOS 5.0 operating system

The `acc` compiler runs under:

- SunOS 4.1.1 (and later) operating system
- A SPARC™ computer, either a server or a workstation
- The OpenWIndows™ 3.0 application development platform.

The SunOS 4.1.1 (and later) operating system is based on the UCB BSD 4.3 operating system.

The `cc` compiler runs under:

- SunOS 5.0 operating system
- A SPARC computer, either a server or a workstation
- The OpenWIndows 3.0 application development platform.

The SunOS 5.0 operating system is based on the System V Release 4 (SVR4) UNIX[1] operating system, and the ONC™ family of published networking protocols and distirbuted services.

Organization of this Book

This book covers the following broad areas:

- introduction and overview of C
- an overview of the compiling process, and an introduction to linking
- the various options available with the `acc` compiler
- the various options available with the `cc` compiler
- the diagnostic, or error, messages you may see when compiling
- data representations
- implementation-specific behavior
- the C programming tools `cscope` and `lint`

In this manual, we do not attempt to teach you how to program in C.

See the manual *Installing SPARCworks and SPARCompiler Software* for instructions on installing the C compiler.

Refer to these other manuals for more information on programming in ANSI C:

C 2.0 Transition Guide
Shows how to port your C code from previous versions of C to ANSI C.

Profiling Tools
Information on various profiling tools.

We recommend two texts for programmers new to the C language:

- Kernighan and Ritchie, *The C Language*, Second Edition, 1988, Prentice-Hall
- Harbison and Steele, *C: A Reference Manual*, Second Edition, 1987, Prentice-Hall.

For implementation-specific details not covered in this book, refer to the *Application Binary Interface* for your machine.

1. UNIX is a registered trademark of UNIX System Laboratories, Inc.

Conventions in this Manual

This manual uses the following conventions:

`Bold face typewriter font`
Indicates commands that you should type in exactly as printed in the manual.

`Regular typewriter font`
Represents what the system prints on your workstation screen, as well as keywords, identifiers, program names, filenames and names of libraries.

Italic font
Indicates variables or parameters that you should replace with an appropriate word or string. It is also used for emphasis.

`$`
Represents your system prompt for a non-privileged user account.

Part 1—ANSI C Overview

Introduction to ANSI C

1.1 Operating Environment

This C compiler runs under two operating environments:

- SunOS™ 4.1.1 (and later) operating system
- SunOS 5.0 operating system

The `acc` compiler runs under:

- SunOS 4.1.1 (and later) operating system
- A SPARC™ computer, either a server or a workstation
- The OpenWIndows™ 3.0 application development platform.

The SunOS 4.1.1 (and later) operating system is based on the UCB BSD 4.3 operating system.

The `cc` compiler runs under:

- SunOS 5.0 operating system
- A SPARC computer, either a server or a workstation
- The OpenWIndows 3.0 application development platform.

The SunOS 5.0 operating system is based on the System V Release 4 (SVR4) UNIX[1] operating system, and the ONC™ family of published networking protocols and distirbuted services.

1.2 C Language

Over the past few years, C has become the worldwide programing language of choice. C was developed on the UNIX operating system and is largely used to code that operating system's kernel. A very large number of UNIX and UNIX-derived applications are written in C.

Chapter 5, *"The Parts of C,"* provides a reference guide to the C language. Here are some features of the language:

- basic data types: characters, integers of various sizes, and floating point numbers;
- derived data types: functions, arrays, pointers, structures, and unions;
- a rich set of operators, including bit-wise operators;
- flow of control: `if`, `if-else`, `switch`, `while`, `do-while`, and `for` statements.

Application programs written in C usually can be transported to other machines without difficulty. Programs written in ANSI standard C (conforming to standards set down by the American National Standards Institute) enjoy an even higher degree of portability.

Programs that require direct interaction with the kernel — for low-level I/O, memory management, interprocess communication, and the like — can be written efficiently in C using the calls to system functions contained in the standard C library, and described in Section 2 of the *SunOS Reference Manual.*

Modular Programming in C

C is a language that lends itself readily to modular programming. It is natural in C to think in terms of functions. And since the functions of a C program can be compiled separately, the next logical step is to put each function, or group of related functions, in its own file. Each file can then be treated as a component, or a module, of your program.

1. UNIX is a registered trademark of UNIX System Laboratories, Inc.

Chapter 2, "Compiling and Linking," describes briefly how to link C programs so that the modules of programs can communicate with each other. What we want to stress here is that coding a program in small pieces eases the job of making changes: you need only recompile the revised modules. It also makes it easier to build programs from code you have written already; as you write functions for one program, you will surely find that many can be picked up for another.

Libraries and Header Files

The standard libraries supplied by the C compilation system contain functions that you can use in your program to perform input/output, string handling, and other high-level operations that are not explicitly provided by the C language. Header files contain definitions and declarations that your program will need if it calls a library function. The functions that perform standard I/O, for example, use the definitions and declarations in the header file `stdio.h`. When you use the line

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

in your program, you ensure that the interface between your program and the standard I/O library agrees with the interface that was used to build the library.

The *C Programmer's Guide* describes some of the more important standard libraries and lists the header files that you need to include in your program if you call a function in those libraries. It also shows you how to use library functions in your program and how to include a header file. You can, of course, create your own libraries and header files.

1.3 Creating an Executable

Chapter 2, "Compiling and Linking," describes the C compilation system, the set of software tools that you use to generate an executable program from C language source files. It contains material that may be of interest to the novice and expert programmer alike.

Additionally, Chapter 2, "Compiling and Linking," details the command line syntax that is used to produce a binary representation of a program — an executable object file. We mentioned earlier that the modules of a C program

can communicate with each other. A symbol declared in one source file can be defined in another, for example. *Link editing* refers to the process whereby the symbol referenced in the first file is connected with the definition in the second. By means of command line options to the `cc` command, you can select either of two link editing models:

- static linking, in which external references are resolved before execution;
- dynamic linking, in which external references are resolved during execution.

Use the `cc` command and its options to control the process in which object files are created from source files, then linked with each other, and with the library functions called in your program.

Chapter 6, "C Error Messages," lists the warning and error messages produced by the C compiler. Check the code examples given in the compiler diagnostics chapter when you need to clarify your understanding of the rules of syntax and semantics summarized in the language chapter. In many cases they'll prove helpful.

1.4 C-Related Programming Tools

There are a number of tools that you can use to aid you in developing, maintaining, and improving your C programs. The two most closely tied to C, `cscope` and `lint`, are described in this manual. Others are described in the *SunOS 5.0 Reference Manual*; some are given detailed treatment in the books *Programming Utilities – SunOS5.0* and *Profiling Tools*.

Program Analysis

`lint`

Checks for code constructs that may cause your C program not to compile, or to execute with unexpected results. `lint` issues every error and warning message produced by the C compiler. It also issues `lint`–*specific* warnings about inconsistencies in definition and use across files and about potential portability problems. The chapter includes a list of these warnings, with examples of source code that would elicit them.

Use lint *to check your program for portability and cross-file consistency, and to assure it will compile.*

`tcov` is also supported in ANSI C. See `tcov` (1) for further information. Profilers are tools that analyze the dynamic behavior of your program: how fast and how often the parts of its code are executed.

`prof`

Reports the amount of time and the percentage of time that was spent executing the parts of a program. It also reports the number of calls to each function and the average execution time of the calls.

`gprof`

In addition to reporting execution times and percentages, like `prof`, `gprof` produces a*call-graph profile* that displays a list of modules that call, and/or are called by, other modules.

`lprof`

A line-by-line frequency profiler. It reports how many times each line of C source code was executed. In that way, it lets you identify the unexecuted and most frequently executed parts of your code. `lprof` is available with SunOS 5.0 only.

`cscope`

An interactive program that locates specified elements of code in C, `lex`, or `yacc` source files. It lets you search and, if you want, edit your source files more efficiently than you could with a typical editor. That's because `cscope` knows about function calls — when a function is being called, when it is doing the calling — and C language identifiers and keywords. `cscope` is available with SunOS 5.0 only.

Use `prof` *and* `lprof` *to identify, and* `cscope` *to rewrite, inefficient lines of code. Use* `cscope` *for any other program-editing task.*

Program Management

`make`

Used to keep track of the dependencies between modules of a program, so that when one module is changed, dependent ones are brought up to date. `make` reads a specification of how the modules of your program depend on each other, and what to do when one of them is modified. When `make` finds a component that has been changed more recently than modules that depend on it, the specified commands — typically to recompile the dependent modules — are passed to the shell for execution.

SCCS

The Source Code Control System, SCCS, is a set of programs that you can use to track evolving versions of files, ordinary text files as well as source files. When a file has been put under control of SCCS, you can specify that only a single copy of any version of it can be retrieved for editing at a time. When the edited file is returned to SCCS, the changes are recorded. That makes it possible to audit the changes and reconstruct the file's earlier versions.

Use make *for any program with multiple files. Use* `SCCS` *to keep track of program versions.*

Program Development

Two system tools were designed to make it easier to build C programs. `lex` and `yacc` generate C language modules that can be useful components of a larger application, in fact, any kind of application that needs to recognize and act on a systematic input.

`lex`

Generates a C language module that performs lexical analysis of an input stream. The lexical analyzer scans the input stream for sequences of characters — tokens — that match regular expressions you specify. When a token is found, an action, which you also specify, is performed.

`yacc`

Generates a C language module that parses tokens that have been passed to it by a lexical analyzer. The parser describes the grammatical form of the tokens according to rules you specify. When a particular grammatical form is found, an action, which again you specify, is taken. The lexical analyzer need not have been generated by `lex`. You could write it in C, with somewhat more effort.

Use `lex` *to create the lexical analyzer, and* `yacc` *the parser, of a user interface.*

Other Advanced Programming Utilities

`m4`

A general-purpose macro processor that can be used to preprocess C and assembly language programs.

Tools for analyzing source code:

`cb`

A C program "beautifier." Formats your source code to make it more readable.

`cflow`

Produces a chart of the external references in C, `lex`, `yacc`, and assembly language files. Use it to check program dependencies.

`ctrace`

Prints out variables as each program statement is executed. Use it to follow the execution of a C program statement by statement.

`cxref`

Analyzes a group of C source files and builds a cross-reference table for the automatic, static, and global symbols in each file. Use it to check program dependencies and to expose program structure.

`indent`

Correctly indents and formats C source files.

Tools for reading and manipulating object files:

`dis`

Dis-assembles object code.

`dump`

Dumps selected parts of object files.

`lorder`

Generates an ordered listing of object files.

`mcs`

Manipulates the sections of an object file.

`nm`

Prints the symbol table of an object file.

`size`

Reports the number of bytes in an object file's sections or loadable segments.

`strip`

Removes symbolic debugging information and symbol tables from an object file.

`unifdef`

Resolves and removes `#ifdef`'d code lines from preprocessor output.

Compiling and Linking

2.1 Compiling and Linking

The C compilation system consists of a compiler, assembler, and link editor. The `cc` command invokes each of these components automatically unless you use command line options to specify otherwise. Before we turn to the `cc` command line syntax, let's look briefly at the four general steps in which an executable C program is created:

1. The preprocessor component of the compiler reads lines in your source files that direct it to replace a name with a token string (`#define`), perhaps conditionally (`#if`, for example).[1] It also accepts directives in your source files to include the contents of a named file in your program (`#include`).

   Included header files for the most part consist of `#define` directives and declarations of external symbols, definitions and declarations that you want to make available to more than one source file.

2. The compiler proper translates the C language code in your source files, which now contain the preprocessed contents of any included header files, into assembly language code.

3. The assembler translates the assembly language code into the machine instructions of the computer your program is to run on. These instructions are stored in object files that correspond to each of your source files. In other

1. The preprocessor is built directly into the compiler (except in `-Xs` mode, where it is called separately).

words, each object file contains a binary representation of the C language code in the corresponding source file. Object files are made up of sections, of which there are usually at least two. The text section consists mainly of program instructions; text sections normally have read and execute, but not write, permissions. Data sections normally have read, write, and execute permissions.

4. The link editor links these object files with each other and with any library functions that you have called in your program, although when it links with the library functions depends on the link editing model you have chosen:

   An archive, or *statically* linked, library

   A statically linked library is a collection of object files each of which contains the code for a function or a group of related functions in the library. When you use a library function in your program, and specify a static linking option on the `cc` command line, a copy of the object file that contains the function is incorporated in your executable at link time.

   A shared object, or *dynamically* linked, library

   A dynamically linked library is a single object file that contains the code for every function in the library. When you call a library function in your program, and specify a dynamic linking option on the `cc` command line, the entire contents of the shared object are mapped into the virtual address space of your process at run time. As its name implies, a shared object contains code that can be used simultaneously by different programs at run time.

We'll discuss briefly these two ways in which libraries are implemented in "Linking Overview" on page 19.

Figure 2-1 shows the organization of the C compilation system. Note that we have omitted discussing the optimizer here because it is optional. (See Chapters 3 and 4).

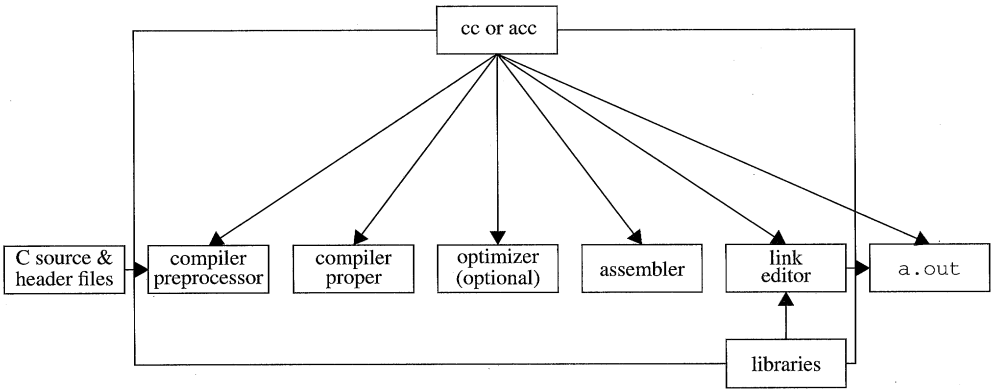

Figure 2-1 Organization of C Compilation System

Here are the specific components, by name (in order):

Table 2-1 Components of C Compilation System

Component	Description	When Used
`cpp`	Preprocessor	`-Xs`
`acomp`	Compiler (preprocessor built in for non-`Xs` modes)	
`basicblk`	Basic block ctr for use with `lprof` (SunOS 5.0 only)	-ql
`iropt`	Code optimizer	`-O` `-x0`*[2-4]* `-xa` `-fast`
`cg`	Code generator	
`inline`	Inline code generator	`.il` file present
`as (cc)`	Assembler	
`fbe (acc)`	Assembler	
`ld`	Linker	

Compiler Command Line Syntax – Basics

Now let's look at how this process works for a C language program called `takeover.c`. Here is the source code for the program:

```
#include <stdio.h>
main(void)
{
     (void) printf("Coelenterates Rule!\n");
     return(0);
}
```

When compiled and executed, the program prints the words *Coelenterates Rule!*

The command to create an executable program from C language source files is `cc`:

```
$ cc takeover.c
```

The source files to be compiled must have names that end in the characters `.c`.

Since we haven't committed any syntactic or semantic errors in our source code, the above command will create an executable program in the file `a.out` in our current directory:

```
$ ls
a.out
takeover.c
```

(Note that no `.o` file is created when you compile a single source file.)

We can execute the program by entering its name after the system prompt:

```
$ a.out
Coelenterates Rule!
```

Since the name `a.out` is only of temporary usefulness, we'll rename the program (or *executable*):

```
$ mv a.out takeover
```

We could also have named the program `takeover` when we compiled it, with the `-o` option to the `cc` command:

```
$ cc -o takeover takeover.c
```

In either case, we execute the program by entering its name after the system prompt:

```
$ takeover
Coelenterates Rule!
```

Now let's look at how the `cc` command controls the four-step process that we described earlier in "Compiling and Linking" on page 9. Using compiler options, we'll break down the compilation process.

When we specify the `-P` option to `cc`, only the preprocessor component of the compiler is invoked:

```
$ cc -P takeover.c
```

The preprocessor's output — the source code plus the preprocessed contents of `stdio.h` — is left in the file `takeover.i` in our current directory:

```
$ ls
takeover.c
takeover.i
```

That output could be useful if, for example, you received a compiler error message for the undefined symbol `a` in the following fragment of erroneous source code:

```
if (i > 4)
{
                    /* declaration follows
     int a;         /* end of declaration */

     a = 4;
}
```

Because the comment on the third line is unterminated (the word *follows* should be followed by */), the compiler would treat the declaration that follows (`int a;`) it as part of the comment. Because the preprocessor removes comments, its output

```
if (i > 4) {
     a = 4;
}
```

would clearly show the effect of the unterminated comment on the declaration.

You can also use the preprocessed output to examine the results of conditional compilation and macro expansion.

If we specify the `-S` option to the `cc` command, only the preprocessor and compiler phases are invoked:

```
$ cc -S takeover.c
```

The output — the assembly language code for the compiled source — is left in the file `takeover.s` in our current directory. That output could be useful if you were writing an assembly language routine and wanted to see how the compiler went about a similar task.

If, finally, we specify the `-c` option to `cc`, all the components but the link editor are invoked:

```
$ cc -c takeover.c
```

The output — the assembled object code for the program — is left in the object file `takeover.o` in our current directory. You would typically want this output when using `make`.

Now we need only enter the command

```
$ cc takeover.o
```

to create the executable object file `a.out`. By default, the link editor arranges for the standard C library function that we have called in our program — `printf()` — to be linked with the executable at run time. In other words, the standard C library is a shared object, at least in the default arrangement we are describing here.

The outputs we have described above are, of course, inputs to the components of the compilation system. They are not the only inputs, however. The link editor, for example, will supply code that runs just before and just after your program to do startup and cleanup tasks. This code is automatically linked with your program only when the link editor is invoked through `cc`. That's why we specified

```
cc takeover.o
```

in the previous example rather than

```
ld takeover.o
```

For similar reasons, you should invoke the assembler through `cc` rather than the assembler (`fbe`):

```
$ cc takeover.s
```

The last line of `takeover.c`

```
return (0);
```

causes the program to terminate gracefully: flushing buffers, closing files, and returning allocated memory to the environment.

When you run this program from a shell, as above, the `return(0);` statement is not needed. However, when you execute it from any environment where the exit status is examined, such as executing from a `make` file, the absence of the statement `return(0);` will cause trouble.

In the `Makefile` example below, the `return(0);` statement has been left out of `takeover.c`.

```
tutorial% cat Makefile
a.out: takeover.c
     cc takeover.c
     a.out
```

Upon execution of the Makefile, you are likely to get the following:

```
tutorial% make
cc takeover.c
a.out
Coelenterates Rule!
*** Error code 1
make: Fatal error: Command failed for target 'a.out'
tutorial%
```

This error occurred because `make` examined `a.out` and discovered that its exit status was undefined and therefore in error. You can use `lint` to detect this error, as shown below.

```
tutorial% lint takeover.c
(6) warning: function has no return statement:  main
function falls off bottom before returning value
     (6) main
tutorial%
```

To correct the program `takeover.c`, add

```
exit(0);
```

or

```
return(0);
```

More generally, if a function is declared with a result type, but ends without returning a result, then the program is in error.

Notice that both `main` and `printf()` are declared to be type `void`. This means that they do not return any value when called . `takeover.c` will compile and run perfectly well without these declarations; however, `lint` will complain about their absence. It's good programming practice to declare the return value of functions, to avoid unexpected return results (an `int` where you expected `a double`, for example).

The compilation process is largely identical if your program is in multiple source files. The only difference is that the default `cc` command line will create object files, as well as the executable object file `a.out`, in your current directory:

```
$ cc file1.c file2.c file3.c
$ ls
a.out
file1.c
file1.o
file2.c
file2.o
file3.c
file3.o
```

What this means is that if one of your source files fails to compile, you need not recompile the others. Suppose, for example, you receive a compiler error diagnostic for `file1.c` in the above command line. Your current directory will look like this:

```
$ ls
file1.c
file2.c
file2.o
file3.c
file3.o
```

That is, compilation proceeds but linking is suppressed. Assuming you have fixed the error, the following command

```
$ cc file1.c file2.o file3.o
```

will create the object file `file1.o` and link it with `file2.o` and `file3.o` to produce the executable program `a.out`. As the example suggests, C source files are compiled separately and independently. To create an executable program, the link editor must connect the definition of a symbol in one source file with external references to it in another.

Note, finally, that not all the `cc` command line options that we have discussed are compiler options. Because, for example, it is the link editor that creates an executable program, the `-o` option — the one you use to give your program a

name other than `a.out` — is actually an `ld` option that is accepted by the `cc` command and passed to the link editor. We'll see further examples of this below. The main reason we mention it is so that you can read about these options on the appropriate manual page.

How C Programs Communicate with the Shell

Information or control data can be passed to a C program as an argument on the command line, which is to say, by the shell. We have already seen, for instance, how you invoke the `cc` command with the names of your source files as arguments:

```
$ cc file1.c file2.c file3.c
```

Note – This section shows examples of invoking the `cc` ("K&R" C compiler). For the ANSI C compiler, use `acc`.

When you execute a C program, command line arguments are made available to the function `main()` in two parameters, an argument count, conventionally called `argc`, and an argument vector, conventionally called `argv`. (Every C program is required to have an entry point named `main`.) `argc` is the number of arguments with which the program was invoked. `argv` is an array of pointers to character strings that contain the arguments, one per string. Since the command name itself is considered to be the first argument, or `argv[0]`, the count is always at least one. Here is the declaration for `main()`:

```
int main(int argc, char *argv[])
```

See the *C 2.0 Libraries Reference Manual* for more information on using these variables.

The shell, which makes arguments available to your program, considers an argument to be any sequence of non-blank characters. Characters enclosed in single quotes (`'abc def'`) or double quotes (`"abc def"`) are passed to the program as one argument even if blanks or tabs are among the characters. You are responsible for error checking and otherwise making sure that the argument received is what your program expects it to be.

C programs exit voluntarily, returning control to the operating system, by returning from `main()` or by calling the `exit()` function. That is, a `return` (*n*) from `main()` is equivalent to the call `exit` (*n*). (Remember that `main()` has type function returning `int`.)

Your program should return a value to the operating system to say whether it completed successfully or not. The value gets passed to the shell, where it becomes the value of the `$?` (Bourne shell) `$status` (C shell) shell variable if you executed your program in the foreground. By convention, a return value of zero denotes success, a non-zero return value means some sort of error occurred. You can use the macros `EXIT_SUCCESS` and `EXIT_FAILURE`, defined in the header file `stdlib.h`, as return values from `main()` or argument values for `exit()`.

In addition to the chapters discussed here, this manual includes appendices on assembly language escapes that use the keyword `asm`, and on mapfiles, a facility for mapping object file input sections to executable file output segments. It also includes a glossary and an index.

Linking Overview

Note – Linking is covered in detail in the manual *Linker and Libraries Manual SunOS 5.0.*

Link editing refers to the process in which a symbol referenced in one module of your program is connected with its definition in another — more concretely, the process by which the symbol `printf()` in our sample source file `takeover.c` is connected with its definition in the standard C library. Whichever link editing model you choose, static or dynamic, the link editor will search each module of your program, including any libraries you have used, for definitions of undefined external symbols in the other modules. If it does not find a definition for a symbol, the link editor will report an error by default, and fail to create an executable program. (Multiply defined symbols are treated differently, however, under each approach.) The principal difference between static and dynamic linking lies in what happens after this search is completed:

- Under static linking, copies of the archive library object files that satisfy still unresolved external references in your program are incorporated in your executable at link time. External references in your program are connected with their definitions — assigned addresses in memory — when the executable is created.
- Under dynamic linking, the contents of a shared object are mapped into the virtual address space of your process at run time. External references in your program are connected with their definitions when the program is executed.

Here are the reasons why you might prefer dynamic to static linking:

- Dynamically linked programs save disk storage and system process memory by sharing library code at run time.
- Dynamically linked code can be fixed or enhanced without having to relink applications that depend on it.

Default Arrangement

We stated earlier that the default `cc` command line

```
$ cc file1.c file2.c file3.c
```

would create object files corresponding to each of your source files, and link them with each other to create an executable program. These object files are called relocatable object files because they contain references to symbols that have not yet been connected with their definitions — have not yet been assigned addresses in memory.

We also suggested that this command line would arrange for the standard C library functions that you have called in your program to be linked with your executable automatically. The standard C library is, in this default arrangement, a shared object called `libc.so`, which means that the functions you have called will be linked with your program at run time. (There are some exceptions. A number of C library functions have been left out of `libc.so` by design. If you use one of these functions in your program, the code for the function will be incorporated in your executable at link time. That is, the function will still be automatically linked with your program, only statically rather than dynamically.) The standard C library contains the system calls

described in Section 2 of the *SunOS Reference Manual*, and the C language functions described in Section 3, Subsections 3C and 3S. See also the *SPARCompiler C Libraries Reference Manual*.

Now let's look at the formal basis for this arrangement:

1. By convention, shared objects, or dynamically linked libraries, are designated by the prefix `lib` and the suffix `.so`; archives, or statically linked libraries, are designated by the prefix `lib` and the suffix `.a`. Then `libc.so` is the shared object version of the standard C library and `libc.a` is the archive version.

2. These conventions are recognized, in turn, by the `-l` option to the `cc` command. That is, the command

```
$ cc file1.c file2.c file3.c -lx
```

   directs the link editor to search the shared object `libx.so` or the archive library `libx.a`. The `cc` command automatically passes `-lc` to the link editor.

3. By default, the link editor chooses the shared object implementation of a library, `libx.so`, in preference to the archive library implementation, `libx.a`, in the same directory.

4. By default, the link editor searches for libraries in the standard places on your system, `/usr/lib` and `/usr/ccs/lib`, in that order. The standard libraries supplied by the compilation system normally are kept in `/usr/lib`.

Adding it up, we can say, more exactly than before, that the default `cc` command line will direct the link editor to search `/usr/lib/libc.so` rather than its archive library counterpart.

`libc.so` is, with one exception, the only shared object library supplied by the C compilation system. (The exception, `libdl.so`, is used with the programming interface to the dynamic linking mechanism described later. Other shared object libraries are supplied with the operating system, and usually are kept in the standard places.) In the next section, we'll show you how to link your program with the archive version of `libc` to avoid the dynamic linking default. Of course, you can link your program with libraries

that perform other tasks as well. Finally, you can create your own shared objects and archive libraries. We'll show you the mechanics of doing that below.

The default arrangement, then, is this: the `cc` command creates and then links relocatable object files to generate an executable program, then arranges for the executable to be linked with the shared C library at run time. If you are satisfied with this arrangement, you need make no other provision for link editing on the `cc` command line.

Linking Summary

By convention, shared objects, or dynamically linked libraries are ,designated by the prefix `lib` and the suffix `.so`; archives, or statically linked libraries, are designated by the prefix `lib` and the suffix `.a`. Then `libc.so` is the shared object version of the standard C library and `libc.a` is the archive version.

1. These conventions are recognized, in turn, by the `-l` option to the `cc` command. That is, `-lx` directs the link editor to search the shared object `libx.so` or the archive library `libx.a`. The `cc` command automatically passes `-lc` to the link editor. In other words, the compilation system arranges for the standard C library to be linked with your program transparently.

2. By default, the link editor chooses the shared object implementation of a library, `libx.so`, in preference to the archive library implementation, `libx.a`, in the same directory.

3. By default, the link editor searches for libraries in the standard places on your system, `/usr/lib` and `/usr/ccs/lib`, in that order. The standard libraries supplied by the compilation system normally are kept in `/usr/lib`.

In this arrangement, then, C programs are dynamically linked with `libc.so` automatically:

```
$ cc file1.c file2.c file3.c
```

To link your program statically with `libc.a`, turn off the dynamic linking default with the `-dn` option:

```
$ cc -dn file1.c file2.c file3.c
```

Specify the `-l` option explicitly to link your program with any other library. If the library is in the standard place, the command

```
$ cc file1.c file2.c file3.c -lx
```

will direct the link editor to search for `libx.so`, then search for `libx.a` in the standard place. Note that the compilation system supplies shared object versions only of `libc` and `libdl`. (Other shared object libraries are supplied with the operating system and usually are kept in the standard places.) Note, too, that, as a rule, it's best to place `-l` at the end of the command line.

If the library is not in the standard place, specify the path of the directory in which it is stored with the `-L` option

```
$ cc -Ldir file1.c file2.c file3.c -lx
```

or the environment variable `LD_LIBRARY_PATH`.

Note – The SunOS 5.0 linker assumes that the LD_LIBRARY_PATH value inthe user's environment, if not semicolon separated, should be interpreted as if the semicolon has been appended. Furthermore, the value cannot be overriden by any other option (such as, -L).

Bourne Shell:

```
$ LD_LIBRARY_PATH=dir; export LD_LIBRARY_PATH
$ cc -Ldir file1.c file2.c file3.c -lx
```

C Shell:

```
% setenv LD_LIBRARY_PATH dir
% cc file1.c file2.c file3.c -lx
```

If the library is a shared object and is not in the standard place, you must also specify the path of the directory in which it is stored with either the environment variable `LD_RUN_PATH` (read by SunOS 5.0 only) at link time, or the environment variable `LD_LIBRARY_PATH` at run time

---

Note – For SunOS 5.0, do not include the `/usr/ccs/lib` directory in the LD_LIBRARY_PATH environment variable. If `/usr/ccs/lib` is included, particularly if placed before `/opt/SUNWspro/SC2.0`, the unbundled compilers will pick up the incorrect `libm.a` from `/usr/ccs/lib`.

Note – For SunOS 5.0, if using `cc` and linking with `FORTRAN` libraries (such as, `cc hello.c -lF77`), set `LD_RUN_PATH` to `/opt/SUNWspro/SC2.0`, or to where ever you have installed the compilers. Alternatively, you can use the `-R` option to `ld`(1) to specify the path.

Bourne Shell:

```
$ LD_RUN_PATH=dir; export LD_RUN_PATH
$ LD_LIBRARY_PATH=dir; export LD_LIBRARY_PATH
```

C Shell:

```
% setenv LD_RUN_PATH dir
% setenv LD_LIBRARY_PATH dir
```

It's best to use an absolute path when you set these environment variables. Note that `LD_LIBRARY_PATH` is read both at link time and at run time.

For SunOS 5.0, to direct the link editor to search `libx.a` where `libx.so` exists in the same directory, turn off the dynamic linking default with the `-dn` option:

```
$ cc -dn -Ldir file1.c file2.c file3.c -lx
```

That command will direct the link editor to search `libc.a` well as `libx.a`. To link your program statically with `libx.a` and dynamically with `libc.so`, use the `-Bstatic` and `-Bdynamic` options to turn dynamic linking off and on:

```
$ cc -Ldir file1.c file2.c file3.c -Bstatic -lx -Bdynamic
```

Files, including libraries, are searched for definitions in the order they are listed on the `cc` command line. The standard C library is always searched last.

acc Compiler Options for SunOS 4.x

This chapter describes the various options available with the C compiler (`acc`).

If you are porting a "K&R" C program to ANSI C, make special note of the sections on `-sys5` and `-X` (compatibility) flags, described later in this chapter. Using them will make the migration to ANSI C easier. Also see the *SPARCompiler C 2.0 Transition Guide*.

The SunOS 4.x operating system is not fully compliant with the ANSI C standard. See Table C-1 andTable C-2 for further details.

3.1 Option Syntax

The syntax of the `acc` command is shown below:

```
tutorial% acc [options] filenames [libraries]...
```

where

- *options* represents one or more of the various options described in this chapter
- *filenames* represent one or more files used in building the executable program

`acc` accepts a list of C source files and object files contained in the list of files specified by *filenames*. The resulting executable code is placed in `a.out`, unless the (-o) option (see below) is used. In that case, the code is placed in the file named by the (-o) option.

`acc` lets you compile and link any combination of the following:

- C source files, with a `.c` suffix
- C preprocessed source files, with a `.i` suffix
- Operating system object-code files, with `.o` suffixes
- Assembler source files, with `.s` suffixes

After linking, `acc` places the linked files, which are now in executable code, into a file named `a.out`, or into the file specified by the -o option.

- *libraries* represents any of a number of standard or user-provided libraries containing functions, macros, and definitions of constants.

Note that unless otherwise specified, options may follow the filename, as in

```
tutorial% acc sourcefilename.c -o outputfilename.
```

3.2 Options

See Table 3-1 on page 40 for a summary of available options.

`-A`*name*`[(`*tokens*`)]`

Associates *name* as a predicate with the specified *tokens* as if by an `#assert` preprocessing directive.

Preassertions:

```
system(unix)
cpu(sparc)
machine(sparc)
```

These preassertions are not valid in `-Xc` mode.

`-a`

This option directs `acc` to insert code to count the number of times the program executes each of its blocks. It then creates a `.d` file with the accumulated execution data for each corresponding `.c` source file. You may then run `tcov`(1) on the source files to generate statistics about the program.

`-a` is not compatible with the `-g` option.

`-B`*binding*

This option specifies whether bindings of libraries for linking are `static` or `dynamic`, indicating whether libraries are non-shared or shared, respectively.

`-C`

This option prevents the C preprocessor from removing comments (except those on preprocessing directive lines).

`-c`

Directs `acc` to suppress linking with `ld(1)` and to produce a `.o` file for each source file. You may explicitly name a single object file using the `-o` option.

`-cg87`

This floating-point code generation option does not exploit features such as the `fsqrts` and `fsqrtd` instructions that are not implemented in hardware on all Sun-4 workstations. It is the default.

`-cg89`

This floating-point code generation option will generate code for any newer Sun-4 that has features like hardware `fsqrts` and `fsqrtd` instructions. Code compiled with `-cg89` should be executed on Sun-4/1xx and Sun-4/2xx systems with Weitek 1164/65 floating-point hardware.

The `-cg87` and `-cg89` options are mutually exclusive. That is, if you compile one procedure with one of these two options, then you should compile all procedures of the program with the same option. Similarly, for a library: compile all procedures in a library with the same `-cg87` or `-cg89` options.

If you are binding an executable, or building a non-shared library, then this consistency is enforced at load time; a message is issued that the link/load failed.

But, if you are building a shared library with `-cg89` and `-pic`, then there is no load-time check for any modules mis-combining `-cg87` and `-cg89` options.

`-D`*name*`[=`*tokens*`]`

Associates *name* with the specified *tokens* as if by a `#define` preprocessing directive. If no `=`*tokens* is specified, the token `1` is supplied.

Predefinitions:

```
sparc
sun
unix
```

These predefinitions are not valid in `-Xc` mode.

`-dalign`

Generates `double` load/store instructions wherever possible for improved performance. Assumes that all double-type data are `double` aligned; `-dalign` should not be used when correct alignment is not assured.

`-dryrun`

This option directs acc to show, but not execute, the commands constructed by the compilation driver.

`-E`

This option runs the source file through the preprocessor only and sends the output to `stdio`.[1] Includes the preprocessor line numbering information. (See also the `-P` option.)

1. The preprocessor is built directly into the compiler (except in `-Xs` mode, where it is called directly).

-fast

This option allows you to select the best combination of compilation options for speed. This should provide close to the maximum performance for most realistic applications.

It is a convenience option, and it chooses the fastest code generation option available on the compile-time hardware (cgx on a Sun-4), the optimization level `-O2`, a set of inline expansion templates, and the `fnonstd` floating-point option.

If you combine `-fast` with other options, the last specification applies. The code generation option, the optimization level and use of inline template files can be overridden by subsequent switches. For example, although the optimization part of `-fast` is `-O2`, the optimization part of `-fast -O1` is `-O1`.

Do not use this option for programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected `SIGFPE` signals.

`-fnonstd`

This option causes non-standard initialization of floating-point arithmetic hardware. By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual.The `-fnonstd` option causes hardware traps to be enabled for floating-point overflow, division by zero, and invalid operations exceptions. These are converted into `SIGFPE` signals, and if the program has no `SIGFPE` handler, it will terminate with a memory dump.

`-fsingle`

(`-Xt` and `-Xs` modes only) Causes the compiler to evaluate `float` expressions as single precision rather than double precision. (This option has no effect if the compiler is used in either `-Xa` or `-Xc` modes, as `float` expressions are already evaluated as single precision.)

`-g`

This option produces additional symbol table information for `dbx`.

Note – Unlike other versions of the C compiler, this version allows the `-O` option to be used with `-g`.

-H

Prints to the standard output, the path name, one per line, of each file included during the current compilation.

The display is indented so as to show which files are included by other files. Here the program `sample.c` includes the files `stdio.h` and `math.h`; `math.h` includes the file `floatingpoint.h`, which itself includes functions that use `ieeefp.h`:

```
$ acc -H sample.c
/usr/include/stdio.h
/usr/include/math.h
     /usr/include/floatingpoint.h
          /usr/include/ieeefp.h
$
```

-help

This option displays information about `acc`.

-I*pathname*

This option adds *pathname* to the list of directories that are searched for `#include` files with relative filenames (those not beginning with slash).

The preprocessor first searches for `#include` files in the directory containing *sourcefile*, then in directories named with `-I` options (if any), and finally, in `/usr/include`. Programs that use system calls, for example, would need to use the file `types.h` as one of their `#include` files. `types.h` contains many type definitions used by common system calls. (See Section 3.3, "Commonly Used Command Line Options," for more details.)

-keeptmp

Causes temporary files created during compilation to be retained instead of deleted automatically.

-L *dir*

Add *dir* to the list of directories searched for libraries by `ld`(1). This option and its arguments are passed to `ld`.

`-l` *library*

This option directs `ld` to link with object library *library*. The ordering of libraries in the command line is important, as symbols are resolved from left to right.

Note – This option must follow the *sourcefile* arguments.

`-libmil`

This option selects the best inline templates for the floating-point option and operating system release available on this system.

`-M`

This option runs only the cpp macro preprocessor on the named C programs, requesting that it generate makefile dependencies and send the result to the standard output (see `make`(1) for details about makefiles and dependencies).

`-misalign`

Generates code to allow loading and storage of misaligned data.

`-native`

This option directs the compiler to compile code targeted for the machine that is doing the compiling. For a Sun-4 that means `-cg87` or `-cg89`.

`-nolibmil`

This option resets `-fast` so that it does not include inline templates. Use it after the `-fast` option. For example: `acc -fast -nolibmil ...`.

`-noqueue`

The `-noqueue` option tells the compiler not to queue this compile request if a license is not available. Under normal circumstances, if no license is available, the compiler waits until one becomes available. With this option, the compiler returns immediately.

`-O[level]`

Equivalent to `-O2`. If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent procedures at the original level specified in the `-O{1,2}` command-line option.

`-o` *outputfile*

This option names the output file *outputfile* (as opposed to the default, `a.out`). *outputfile* must have the appropriate suffix for the type of file to be produced by the compilation. *outputfile* cannot be the same as *sourcefile*, since `acc` will not overwrite the source file. This option and its arguments are passed to `ld`(1).

`-P`

This option runs the source file through the C preprocessor only. It then puts the output in a file with a `.i` suffix. Unlike `-E`, it does not include preprocessor-type line number information in the output. (See also the `-E` option.)

`-p`

This option prepares the object code to collect data for profiling with `prof`(1). `-p` invokes a run-time recording mechanism that produces a `mon.out` file at normal termination. See *Profiling Tools* for more on `prof`.

`-pg`

This option prepares the object code to collect data for profiling with `gprof`(1). It invokes a run-time recording mechanism that produces a `gmon.out` file at normal termination.

`-PIC`

This option produces position-independent code. Each reference to a global datum is generated as a de-reference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

`-PIC` lets the global offset table span the range of 32-bit addresses in those rare cases where there are too many global data objects for `-pic`.

`-pic`

This option produces position-independent code. It is similar to `-PIC`, but the size of the global offset table is limited to 8K.

There are two nominal performance costs with `-pic` and `-PIC`, namely:

- A routine compiled with either `-pic` or `-PIC` executes a few extra instructions upon entry (in order to set a register to point at a table (`_GLOBAL_OFFSET_TABLE_`) used for accessing a shared library's global or static variables.
- Each access to a global or static variable involves an extra indirect memory reference through `_GLOBAL_OFFSET_TABLE_`. (If the compile is done with `-PIC`, there are an additional two instructions per global/static memory reference.)

When considering the above costs, one should remember that the use of `-pic` and `-PIC` can significantly reduce system memory requirements, due to the effect of library code sharing. Every page of code in a shared library compiled `-pic` or `-PIC` can be shared by every process that uses the library. If a page of code in a shared library contains even a single non-pic (i.e., absolute) memory reference, the page becomes nonsharable, and a copy of the page must be created each time a program using the library is executed.

The easiest way to tell whether or not a `.o` file has been compiled with `-pic` or `-PIC` is with the `nm` command:

```
tutorial% nm file.o | grep __GLOBAL_OFFSET_TABLE_
U __GLOBAL_OFFSET_TABLE_
tutorial%
```

A `.o` file containing position-independent code will contain an unresolved external reference to `__GLOBAL_OFFSET_TABLE_` (indicated by the letter *U*).

To determine whether to use `-pic` or `-PIC`, use `nm` to identify the number of distinct global/static variables used or defined in the library. If the size of `_GLOBAL_OFFSET_TABLE_` is under 8192 bytes, you may use `-pic`. Otherwise, you must use `-PIC`.

`-Qdir` or `-qdir` *directory*

This option allows you to search for compiler components in *directory X*.

`-Qoption` or `-qoption` *prog opt*

This option passes the option *opt* to the compiler phase *prog*. The option must be appropriate to that program and may begin with a minus sign. *prog* can be one of `as`(1), `cpp`(1), `inline`(1), or `ld`(1).

`-Qpath` or `-qpath` *pathname*

This option inserts a directory *pathname* into the search path used to locate compiler components. This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver, for example `*crt*.o` and `bb_link.o`. This lets you choose whether or not to use default versions of programs invoked during compilation.

`-Qproduce` or `-qproduce` *sourcetype*

This option causes `acc` to produce source code of the type *sourcetype*. *sourcetype* can be one of the following:

`.c` C source.

`.i` Preprocessed C source from `cpp`.

`.o` Object file from `as`.

`.s` Assembler source (from `acomp`, or `inline`(1)).

`-R`

This option directs `acc` to merge the data segment with the text segment for `as`(1). Data initialized in the object file produced by this compilation is read-only, and (unless linked with `ld -N`) is shared between processes. This option is ignored when `-g` is used.

`-S`

This option directs `acc` to produce an assembly source file but not to assemble the program.

`-s`

Removes all symbolic debugging information from the output object file. Passed to `ld`(1).

`-sb`

This option generates and compiles extra symbol table information for the SourceBrowser.

`-sbfast`

This option generates, but does not compile, extra symbol table information for the SourceBrowser.

`-strconst`

This option inserts string literals into the text segment instead of the data segment.

`-sys5`

This option adds the SystemV header files and libraries to the compiler directory search paths. See Table C-1 and Table C-2 for further details.

`-temp=` *dir*

This option sets the directory to contain temporary files generated during the compilation process to be *dir*.

`-time`

This option directs `acc` to report execution times for the various compilation passes.

`-U`*name*

This option removes any initial definition of the preprocessor symbol *name*. This option is the inverse of the `-D` option. Multiple `-U` options may be given.

`-V`

This option directs `acc` to print the name and version ID of each pass as the compiler executes.

`-v`

Verbose. Print the version number of the compiler and the name of each program it executes.

`-vc`

This option directs the compiler to perform stricter semantic checks and to enable other lint-like checks. For example, the code

```
#include <stdio.h>
main(void)
{
     printf("Solipsism isn't for everybody.\n");
}
```

will compile and execute without problem. With `-vc`, it still compiles; however, the compiler displays this warning:

```
"solipsism.c", line 5: warning: function has no return
   statement: main
```

Note that `-vc` does not give all the warnings that `lint`(1) does. (Try running the above example through `lint`.)

See Chapter 6, *"C Error Messages,"* for an explanation of the compiler error messages.

`-w`

This option directs `acc` to not print warnings.

The following `-X` (note case) options provide varying degrees of compliance to the ANSI C standard. `-Xt` is the default mode.

`-Xa`

(*a = ANSI*) ANSI C plus K&R C compatibility extensions, *with* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the ANSI C interpretation.

`-Xc`

(*c = conformance*) Maximally conformant ANSI C, without K&R C compatibility extensions. The compiler will reject programs that use non-ANSI C constructs.

`-Xs`

(*s = senescent*) The compiled language includes all features compatible with (pre-ANSI) K&R C. The computer warns about all language constructs that have differing behavior between ANSI C and the old K&R C.

`-Xt`

(*t = transition*) ANSI C plus K&R C compatibility extensions, *without* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the K&R C interpretation. This is the default compiler mode.

`-xlicinfo`

The `-xlicinfo` option returns information about the licensing system. In particular, it returns the name of the license server and the userids of users who have licenses checked out. When you use this option, the compiler is not invoked and a license is not checked out.

Summary of Compiler Options

Table 3-1 Summary of Compiler Options (Sheet 1 of 3)

Option or Flag	Description
`-A`*name [(tokens)]*	Preprocessor predicate assertion
`-a`	Count number of times a program executes each of its blocks
`-B`*binding*	Specify binding type (dynamic or static)
`-C`	Preprocessor comments left in
`-c`	Produce `.o` file but do not actually do linking
`-cg87`	Generates floating-point code; `fsqrts` nad `fsqrtd` not implemented in hardware
`-cg89`	Generates floating-point code; `fsqrts` nad `fsqrtd` implemented in hardware
`-D`*name*[=*token*]	Associate *name* with *token* as if by `#define`
`-dalign`	Assume doubles are doubleword aligned
`-dryrun`	Show constructed command, but do not execute
`-E`	Run source through preprocessor only
`-fast`	Options for best performance
`-fnonstd`	Non-standard initialization of floating-point hardware
`-fsingle`	Use single-precision arithmetic (`-Xt` and `-Xs` modes only)
`-g`	Generate info for `dbx`
`-H`	Print paths of included files during compilation
`-help`	Display information about `acc`
`-I`*pathname*	Add *dir* to include search path
`-keeptmp`	Keep temporary files
`-Ldir`	Add *dir* to `ld` library path
`-llibrary`	Link object library (for `ld`)
`-libmil`	Select best inline templates for floating-point
`-M`	Run macro processor only; generate makefile dependencies
`-misalign`	Allow loading and storage of misaligned data
`-native`	Target code for machine doing the compiling

Table 3-1 Summary of Compiler Options (Sheet 2 of 3)

Option or Flag	Description
`-nolibmil`	Reset -fast; do not include inline templates
`-noqueue`	Do not queue compiler request if license is unavailable
`-o` *file*	Set name of output file
`-O`	Generate optimized code (equivalent to `-xO2`)
`-P`	Run source through preprocessor only; send output to `.i` file
`-p`	Collect data for `prof`
`-pg`	Collect data for gprof
`-PIC`	Produce position independent code
`-pic`	Like `-PIC`, but with a smaller global offset table
`-Qdir` or `-qdir` *dir*	Search for compiler components in *dir x*
`-Qoption` or `-qoption` *prog opt*	Pass option *opt* to compiler phase *prog*
`-Qpath` or `-qpath` *pathname*	Insert directory *pathname* into compiler component search path
`-Qproduce or -qproduce` *sourcetype*	Produce source code of type *sourcetype*
`-R`	Merge data segment with text segment for assembler
`-S`	Product `.s` file only (do not assemble or link)
`-s`	strip (4.1); pass to `ld` (5.0)
`-sb`	Generate and compile symbol table information for SourceBrowser
`-sbfast`	Generate, but do not compile, symbol table information for Source-Browser
`-strconst`	Insert string literals into text segment rather than data segment
`-sys5`	Add SystemV header files and libraries to directory search path
`-temp=` *dir*	Sets the directory *dir* to contain compiler generated temporary files
`-time`	Report execution times for various compilation passes
`-U`*name*	Undefine preprocessor symbol *name* as if by `#undef`
`-V`	Report versions of invoked programs
`-v`	Print compiler version no. and name of programs executed

Table 3-1 Summary of Compiler Options (Sheet 3 of 3)

Option or Flag	Description
`-vc`	Impose stricter semantic checks and enable other lint-like checks
`-w`	Do not print warnings
`-Xa`	(a = ANSI) Compatibility options (ANSI, conformant, K&R C, transition)
`-Xc`	(c = conformance) Maximally conformant ANSI C, without K&R C compatibility extensions
`-Xs`	(s = senescent) Compiled language includes all features compatible with (pre-ANSI) K&R C
`-Xt`	(t = transition) ANSI C plus K&R compatibility extensions, without semantic changes required by ANSI C
`-xlicinfo`	Returns information about the licensing system

3.3 Commonly Used Command Line Options

Searching for a Header File

Recall that the first line of our sample program was

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

The format of that directive is the one you should use to include any of the standard header files that are supplied with the C compilation system. The angle brackets (<>) tell the preprocessor to search for the header file in the standard place for header files on your system, usually the `/usr/include` directory.

The format is different for header files that you have stored in your own directories:

```
#include "header.h"
```

The quotation marks (`" "`) tell the preprocessor to search for `header.h` first in the directory of the file containing the `#include` line, which will usually be your current directory, then in the standard place.

If your header file is not in the current directory, specify the path of the directory in which it is stored with the `-I` option to `acc`. Suppose, for instance, that you have included both `stdio.h` and `header.h` in the source file `mycode.c`:

```
#include <stdio.h>
#include "header.h"
```

Suppose further that `header.h` is stored in the directory `. . ./defs`. The command

```
$ acc -I../defs mycode.c
```

will direct the preprocessor to search for `header.h` first in the current directory, then in the directory `. . ./defs`, and finally in the standard place. It will also direct the preprocessor to search for `stdio.h` first in `. . ./defs`, then

in the standard place — the difference being that the current directory is searched only for header files whose name you have enclosed in quotation marks.

You can specify the `-I` option more than once on the `acc` command line. The preprocessor will search the specified directories in the order they appear on the command line. Needless to say, you can specify multiple options to `acc` on the same command line:

```
$ acc -o prog -I../defs mycode.c
```

Preparing Your Program for Symbolic Debugging

When you specify the `-g` option to `acc`,

```
$ acc -g mycode.c
```

you arrange for the compiler to generate information about program variables and statements that will be used by the symbolic debugger `dbx`. The information supplied to `dbx` will allow you to use the symbolic debugger to trace function calls, display the values of variables, set breakpoints, and so on.

Note – Both the -o and -g options support the debugging of optimized code. For detailed information, see the discussion in *Debugging a Program.*

Preparing Your Program for Profiling

The various profilers for optimizing your source code are described briefly in Chapter 1, "Introduction to ANSI C," and extensively in *Profiling Tools.*

To use the profilers that are supplied with the C compilation system, you must do two things:

1. **Compile and link your program with a profiling option:**

```
$ acc -pg -o prog mycode.c      (for gprof)
$ acc -a -o prog mycode.c       (for tcov)
```

2. **Run the profiled program:**

```
$ prog
```

At the end of execution, data about your program's run-time behavior is written to a file in your current directory:

```
$ ls
gmon.out                        (for gprof)
mon.out                         (for prof)
mycode.c
mycode.d                        (for tcov)
```

3. **Run the profiler:**

```
$ gprof prog > output.file
$ tcov mycode.c                 (produces mycode.tcov file)
$ prof prog > output.file
```

The files are inputs to the profilers.

See the *Profiling Tools* manual for more information on `gprof`(1), `tcov`(1), and `prof`(1).

Non-Standard Floating Point

IEEE 754 floating-point default arithmetic is "nonstop" and underflows are "gradual." What do we mean by these terms?

This explanation is by necessity rather over-simplified. See the *Numerical Computation Guide* for more rigorous descriptions.

Nonstop means that execution doesn't halt on things like division by zero, floating-point overflow, or invalid operation exceptions. For example, consider the following, where `x` is zero and `y` is positive:

```
z = y / x;
```

By default, `x` gets set to the value `+Inf`, and execution continues. With the `-fnonstd` option, however, this code causes an ungraceful exit (say, a core dump).

Here's how gradual underflow works. Suppose you have the following code:

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
     x = x / 10;
```

The first time through the loop, `x` is set to 1; the second time through, to 0.1; the third time through, to 0.01; and so on. Eventually, `x` will reach the lower limit of the machine's capacity to represent its value. What happens the next time the loop runs?

Let's say that the smallest number characterizable is

```
1.234567e-38
```

The next time the loop runs, the number is modified by "stealing" from the mantissa and "giving" to the exponent:

```
1.23456e-39
```

and, subsequently,

```
1.2345e-40
```

and so on. This is know as "gradual underflow" and it's the default behavior. In non-standard behavior, none of this "stealing" goes on; typically, `x` is simply set to zero.

cc Compiler Options for SunOS 5.0

This chapter describes the various options available with the C compiler (`cc`).

If you are porting a "K&R" C program to ANSI C, make special note of the section on `-X` (compatibility) flags described later in this chapter. Using them will make the migration to ANSI C easier. And see also the *C 2.0 Transition Guide*.

4.1 Option Syntax

The syntax of the `cc` command is shown below:

```
tutorial% cc [options] filenames [libraries]...
```

where

- *options* represents one or more of the various options described in this chapter
- *filenames* represent one or more files used in building the executable program

  `cc` accepts a list of C source files and object files contained in the list of files specified by *filenames*. The resulting executable code is placed in `a.out`, unless the (-o) option (see below) is used. In that case, the code is placed in the file named by the (-o) option.

  `cc` lets you compile and link any combination of the following:

- C source files, with a `.c` suffix
- C preprocessed source files, with a `.i` suffix
- Operating system object-code files, with `.o` suffixes
- Assembler source files, with `.s` suffixes

After linking, `cc` places the linked files, which are now in executable code, into a file named `a.out`, or into the file specified by the -o option.

- *libraries* represents any of a number of standard or user-provided libraries containing functions, macros, and definitions of constants.

Note that unless otherwise specified, options may follow the filename, as in

```
tutorial% cc sourcefilename.c -o outputfilename.
```

4.2 Options

See Table 4-1 on page 62 for a summary of available options.

`-#`

Causes the compiler to work in *verbose mode*, showing each component as it is invoked.

`-###`

Shows each component as it is invoked, but does not actually execute it.

`-A`*name*`[(`*tokens*`)]`

Associates *name* as a predicate with the specified *tokens* as if by an `#assert` preprocessing directive.

Preassertions:

```
system(unix)
cpu(sparc)
machine(sparc)
```

These preassertions are not valid in `-Xc` mode.

`-B`*binding*

This option specifies whether bindings of libraries for linking are `static` or `dynamic`, indicating whether libraries are non-shared or shared, respectively.

`-C`

This option prevents the C preprocessor from removing comments (except those on preprocessing directive lines).

`-c`

Directs `cc` to suppress linking with `ld(1)` and to produce a `.o` file for each source file. You may explicitly name a single object file using the `-o` option.

`-D`*name*`[=`*tokens*`]`

Associates *name* with the specified *tokens* as if by a `#define` preprocessing directive. If no `=`*tokens* is specified, the token `1` is supplied.

Predefinitions:

```
sparc
sun
unix
```

These predefinitions are not valid in `-Xc` mode.

`-d`*c*

c can be either `y` or `n`.

- `-dy` specifies dynamic linking, which is the default, in the link editor.
- `-dn` specifies static linking in the link editor.

This option and its arguments are passed to `ld`(1).

`-dalign`

Generates `double` load/store instructions wherever possible for improved performance. Assumes that all double-type data are `double` aligned; `-dalign` should not be used when correct alignment is not assured.

`-E`

This option runs the source file through the preprocessor only and sends the output to `stdio`.[1] Includes the preprocessor line numbering information. (See also the `-P` option.)

`-F`*option*

Reserved for future floating-point options.

`-fast`

This option allows you to select the best combination of compilation options for speed. This should provide close to the maximum performance for most realistic applications.

It is a convenience option, and it chooses the fastest code generation option available on the compile-time hardware, the optimization level `-xO2`, a set of inline expansion templates, and the `fnonstd` floating-point option. It also adds `-lm` to link in the math library.

If you combine `-fast` with other options, the last specification applies. The code generation option, the optimization level and use of inline template files can be overridden by subsequent switches. For example, although the optimization part of `-fast` is `-xO2`, the optimization part of `-fast -xO1` is `-xO1`.

Do not use this option for programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected `SIGFPE` signals.

`-flags`

Prints a one-line summary of each option.

`-fnonstd`

This option causes non-standard initialization of floating-point arithmetic hardware. By default, IEEE 754 floating-point arithmetic is nonstop, and underflows are gradual. (See "Non-Standard Floating Point" on page 67 for a further explanation.) The `-fnonstd` option causes hardware traps to be

1. The preprocessor is built directly into the compiler (except in `-Xs` mode, where it is called directly).

enabled for floating-point overflow, division by zero, and invalid operations exceptions. These are converted into SIGFPE signals, and if the program has no SIGFPE handler, it will terminate with a memory dump.

-fnonstd also causes the math library to be linked in, by passing -lm to the linker.

-fsingle

(-Xt and -Xs modes only) Causes the compiler to evaluate float expressions as single precision rather than double precision. (This option has no effect if the compiler is used in either -Xa or -Xc modes, as float expressions are already evaluated as single precision.)

-G

Used to direct the link editor to produce a shared object rather than a dynamically linked executable. This option is passed to ld(1). It cannot be used with the-dn option.

-g

This option produces additional symbol table information for dbx.

Note – Unlike other versions of the C compiler, this version allows the -O option to be used with -g.

-H

Prints to the standard output, the path name, one per line, of each file included during the current compilation.

The display is indented so as to show which files are included by other files. Here the program sample.c includes the files stdio.h and math.h; math.h includes the file floatingpoint.h, which itself includes functions that use ieeefp.h:

```
$ cc -H sample.c
/usr/include/stdio.h
/usr/include/math.h
     /usr/include/floatingpoint.h
          /usr/include/ieeefp.h
$
```

`-h` *name*

This option assigns a name to a shared dynamic library as a way to have different versions of a library. In general, the *name* after `-h` should be the same as the filename given after the `-o` option. (The space between `-h` and *name* is optional.)

The loader assigns the specified *name* to the library and records the name in the library file as the *intrinsic* name of the library. If there is no `-h`*name* option, then no intrinsic name is recorded in the library file.

When the run-time linker loads the library into an executable file, it copies the intrinsic name from the library file into the executable, into a list of needed shared library files. (Every executable has such a list.) If there is no intrinsic name of a shared library, then the linker copies the path of the shared library file instead.

Here's how you'd make and use one version of a shared library:

```
$ ld -G -o libxyz.1 -h libxyz.1 ...    (create shared library)
$ ln libxyz.1 libxyz.so                (link libxyz.so to libxyz.1)
$ cc -o verA -lxyz . . .               (executable verA needs libxyz.1)
```

Here's how you'd make and use a different version of the library:

```
$ ld -G -o libxyz.2 -h libxyz.2 ...    (create shared library)
$ rm libxyz.so                         (remove old link)
$ ln libxyz.2 libxyz.so                (link libxyz.so to libxyz.2)
$ cc -o verB -lxyz . . .               (executable verB needs libxyz.2)
```

`-I`*pathname*

This option adds *pathname* to the list of directories that are searched for `#include` files with relative filenames (those not beginning with slash).

The preprocessor first searches for `#include` files in the directory containing *sourcefile*, then in directories named with `-I` options (if any), and finally, in `/usr/include`.

`-i`

This option tells the compiler to ignore any `LD_LIBRARY_PATH` setting.

`-KPIC`

This option produces position-independent code. Each reference to a global datum is generated as a de-reference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table.

`-KPIC` lets the global offset table span the range of 32-bit addresses in those rare cases where there are too many global data objects for `-Kpic`.

`-Kpic`

This option produces position-independent code. It is similar to `-KPIC`, but the size of the global offset table is limited to 8K.

There are two nominal performance costs with `-kpic` and `-KPIC`, namely:

- A routine compiled with either `-kpic` or `-KPIC` executes a few extra instructions upon entry (in order to set a register to point at a table (`_GLOBAL_OFFSET_TABLE_`) used for accessing a shared library's global or static variables.
- Each access to a global or static variable involves an extra indirect memory reference through `_GLOBAL_OFFSET_TABLE_`. (If the compile is done with `-KPIC`, there are an additional two instructions per global/static memory reference.)

When considering the above costs, one should remember that the use of `-kpic` and `-KPIC` can significantly reduce system memory requirements, due to the effect of library code sharing. Every page of code in a shared library compiled `-kpic` or `-KPIC` can be shared by every process that uses the library. If a page of code in a shared library contains even a single non-pic (i.e., absolute) memory reference, the page becomes nonsharable, and a copy of the page must be created each time a program using the library is executed.

The easiest way to tell whether or not a `.o` file has been compiled with `-kpic` or `-KPIC` is with the `nm` command:

```
tutorial% nm file.o | grep __GLOBAL_OFFSET_TABLE_
U __GLOBAL_OFFSET_TABLE_
tutorial%
```

An `.o` file containing position-independent code will contain an unresolved external reference to `__GLOBAL_OFFSET_TABLE_` (indicated by the letter *U*).

To determine whether to use `-kpic` or `-KPIC`, use `nm` to identify the number of distinct global/static variables used or defined in the library. If the size of `_GLOBAL_OFFSET_TABLE_` is under 8192 bytes, you may use `-kpic`. Otherwise, you must use `-KPIC`.

`-keeptmp`

Causes temporary files created during compilation to be retained instead of deleted automatically.

`cc` normally creates temporary files in the directory `/var/tmp`. You may specify another directory by setting the environment variable `TMPDIR` to the directory of your choice. (If `TMPDIR` isn't a valid directory, `cc` will use `/var/tmp`.)

Bourne Shell:

```
$ TMPDIR=dir; export TMPDIR
```

C Shell:

```
% setenv TMPDIR dir
```

`-L`*dir*

Add *dir* to the list of directories searched for libraries by `ld`(1). This option and its arguments are passed to `ld`.

`-l`*library*

This option directs `ld` to link with object library *library*. The ordering of libraries in the command line is important, as symbols are resolved from left to right.

Note – This option must follow the *sourcefile* arguments.

`-misalign`

Generates code to allow loading and storage of ccmisaligned data.

`-noqueue`

The `-noqueue` option tells the compiler not to queue this compile request if a license is not available. Under normal circumstances, if no license is available, the compiler waits until one becomes available. With this option, the compiler returns immediately.

`-O`

Equivalent to `-xO2`.

`-o` *outputfile*

This option names the output file *outputfile* (as opposed to the default, `a.out`). *outputfile* cannot be the same as *sourcefile*, since `cc` will not overwrite the source file. This option and its arguments are passed to `ld`(1).

`-P`

This option runs the source file through the C preprocessor only. It then puts the output in a file with a `.i` suffix. Unlike `-E`, it does not include preprocessor-type line number information in the output. (See also the `-E` option.)

`-p`

This option prepares the object code to collect data for profiling with `prof`(1). `-p` invokes a run-time recording mechanism that produces a `mon.out` file at normal termination. See *Profiling Tools* for more on `prof`.

`-Qc`

c can be either `y` or `n`. `-Qy` is the default.

If *c* is `y`, identification information about each invoked compilation tool will be added to the output files. This can be useful for software administration.

`-Qn` suppresses this information.

`-q`*c*

c can be either `1` or `p`.

`-ql` causes the invocation of the basic block analyzer and arranges for the production of code that counts the number of times each source line is executed. A listing of these counts can be generated by use of `lprof`(1).

`-qp` is a synonym for `-p`. `-q` cannot be used with either `-O` or `-xO` options. See *Profiling Tools* for more on `lprof`.

`-R`*dir*`[:dir]`

This option specifies the library search path for the dynamic linker.

`-S`

This option directs `cc` to produce an assembly source file but not to assemble the program.

`-s`

Removes all symbolic debugging information from the output object file. Passed to `ld`(1).

`-U`*name*

This option removes any initial definition of the preprocessor symbol *name*. This option is the inverse of the `-D` option. Multiple `-U` options may be given.

`-V`

This option directs `cc` to print the name and version ID of each pass as the compiler executes.

`-v`

This option directs the compiler to perform stricter semantic checks and to enable other lint-like checks. For example, the code

```
#include <stdio.h>
main(void)
{
     printf("Solipsism isn't for everybody.\n");
}
```

will compile and execute without problem. With `-v`, it still compiles; however, the compiler displays this warning:

```
"solipsism.c", line 5:  warning:  function has no return
  statement: main
```

Note that `-v` does not give all the warnings that `lint`(1) does. (Try running the above example through `lint`.)

See Chapter 6, "*C Error Messages*," for an explanation of the compiler error messages.

`-W`*tool*,*arg*$_1$[*arg*$_2$]

Hands off the argument(s) *arg*$_i$ each as a separate argument to *tool*. Each argument must be separated from the preceding by only a comma. (A comma can be part of an argument by escaping it by an immediately preceding backslash (`\`) character.) *tool* can be one of the following:

`a`	assembler (fbe)
`b`	basic block analyzer (basicblk)
`c`	C code generator (cg)
`i`	inliner (inline)
`l`	link editor (ld)
`p`	preprocessor (cpp)
`0`	compiler (acomp)
`2`	optimizer (iropt)

`-w`

This option directs `cc` to not print warnings.

The following `-X` (note case) options provide varying degrees of compliance to the ANSI C standard. `-Xt` is the default mode.

`-Xa`

(*a = ANSI*) ANSI C plus K&R C compatibility extensions, *with* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the ANSI C interpretation.

`-Xc`

(*c = conformance*) Maximally conformant ANSI C, without K&R C compatibility extensions. The compiler will reject programs that use non-ANSI C constructs.

`-Xs`

(*s = senescent*) The compiled language includes all features compatible with (pre-ANSI) K&R C. The computer warns about all language constructs that have differing behavior between ANSI C and the old K&R C.

`-Xt`

(*t = transition*) ANSI C plus K&R C compatibility extensions, *without* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the K&R C interpretation. This is the default compiler mode.

`-xa`

Inserts code to count how many times each basic block is executed. Invokes a run-time recording mechanism that creates a `.d` file for every `.c` file (at normal termination). The `.d` file accumulates execution data for the corresponding source file. `tcov`(1) can then be run on the source file to generate statistics about the program. Since this option entails some optimization, it is incompatible with `-g`. See *Profiling Tools* for more on `tcov`.

`-xF`

Produces code that can be re-ordered at the function level. Each function in the file is placed in a separate section; for example, functions `foo()` and `bar()` will be placed in the sections `.text%foo` and `.text%bar`, respectively. Function ordering in the executable can be controlled by using `-xF` in conjunction with the `-M` option to `ld`(1).

`-xlibmil`

Includes inline expansion templates for `libm`.

`-xlicinfo`

The `-xlicinfo` option returns information about the licensing system. In particular, it returns he name of the license server and the userids of users who have licenses checked out. When you give this option, the compiler is not invoked and a license is not checked out.

`-xM`

This option runs only the macro preprocessor (`/usr/ccs/bin/cpp`) on the named C programs, requesting that it generate makefile dependencies and send the result to the standard output (see `make`(1) for details about makefiles and dependencies).

`-xnolibmil`

This option resets `-fast` so that it does *not* include inline templates. Use it after the `-fast` option. For example:

```
cc -fast -xnolibmil....
```

`-xO[`*level*`]`

This option directs `cc` to optimize the object code. `-xO[`*level*`]` may be combined with`-g`, but not with `-a`.

Note – Unlike other versions of the C compiler, this version allows the `-O` option to be used with `-g`.

---

level can be one of the following:

1

Do postpass assembly-level optimization only.

2

Do global optimization before code generation, including loop optimizations, common subexpression elimination, copy propagation, and automatic register allocation. `-xO2` does not optimize references to or definitions of external or indirect variables.

3

Same as `-xO2`, but do loop unrolling and optimize uses and definitions of external variables. `-xO3` does not trace the effects of pointer assignments.

4

Same as `-xO3`, but trace the effects of pointer assignments, gather aliasing information, and do parameter propagation, and perform inlining.

Note – If you use `-O` without specifying the *level*, it is equivalent to using `-xO2`.

Neither `-xO3` nor `-xO4` should be used when compiling either device drivers or programs that modify external variables from within signal handlers. Also, levels `-xO3` and `-xO4` may increase the size of executables; when optimizing for size, use `-xO2`.

If the optimizer runs out of memory, it tries to recover by retrying the current procedure at a lower level of optimization and resumes subsequent procedures at the original level specified in the `-xO{1,2}` command-line option.

`-xpg`

This option prepares the object code to collect data for profiling with `gprof`(1). It invokes a run-time recording mechanism that produces a `gmon.out` file at normal termination. See *Profiling Tools* for more on `gprof`.

`-xs`

This option disables *autoload* for `dbx`. This is in case you cannot keep the.o files around. This passes the `-s` option to the assembler and the linker.

No Autoload: This is the older way of loading symbol tables.

- Place all symbol tables for `dbx` in the executable file.
- The linker links more slowly and `dbx` initializes more slowly.
- If you move the executables to another directory, then to use `dbx` you must move the source files, but you do not need to move the object (`.o`) files.

Autoload: This is the newer (and default) way of loading symbol tables.

- Distribute this information in the `.o` files so that `dbx` loads the symbol table information only if and when it is needed.
- The linker links faster and `dbx` initializes faster.
- If you move the executables to another directory, then to use `dbx` you must move both the source files and the object (`.o`) files.

`-xsb`

This option generates extra symbol table information for the SourceBrowser.

`-xsbfast`

Same as `-xsb`, but does not actually compile.

`-xstrconst`

This option inserts string literals into the read-only data segment instead of the default data segment.

`-Y`*item, dir*

Specify a new directory *dir* for the location of *item*. *item* can consist of any of the characters representing tools that are listed under the `-W` option, or it may be any of the following characters representing directories containing special files:

`I`	directory searched last for include files: `INCDIR` (see `-I`)
`P`	New default directories for finding libraries; *dir* in this case is a colon-separated path list.
`S`	directory containing the start-up object files: `LIBDIR`

If the location of a tool is being specified, then the new path name for the tool will be *dirtool*. If more than one `-Y` option is applied to any one item, then the last occurrence holds.

Summary of `cc` *Options*

Table 4-1 Summary of `cc` Options (Sheet 1 of 3)

Option or Flag	Description
`-#`	Verbose mode
`-###`	Show components, but do not execute
`-A`*symbol*	Preprocessor predicate assertion
`-B`*binding*	Specify binding type (dynamic or static)
`-C`	Preprocessor comments left in
`-c`	Produce `.o` file but do not actually do linking
`-D`*name*`[=`*token*`]`	Associate *name* with *token* as if by `#define`
`-d[y|n]`	Dynamic linking [yes | no]
`-dalign`	Assume doubles are doubleword aligned
`-E`	Run source through preprocessor only
`-F`	Reserved for future floating-point optimization directives
`-fast`	Options for best performance
`-flags`	Print summary of compiler options
`-fnonstd`	Non-standard initialization of floating-point hardware
`-fsingle`	Use single-precision arithmetic (`-Xt` and `-Xs` modes only)
`-G`	Like `-dy`, but no `crt1.o` is linked
`-g`	Generate info for `dbx`
`-H`	Print paths of included files during compilation
`-h`	Name a shared dynamic library
`-I`*dir*	Add *dir* to include path
`-i`	Ignore any `LD_LIBRARY_PATH setting`
`-KPIC`	Produce position independent code
`-Kpic`	Like `KPIC`, but with a smaller global offset table
`-keeptmp`	Keep temporary files
`-L`*dir*	Add *dir* to `ld` library path
`-l`*dir*	Read object library (for `ld`)

Table 4-1 Summary of `cc` Options (Sheet 2 of 3)

Option or Flag	Description
`-misalign`	Allow loading and storage of misaligned data
`-noqueue`	Don't queue license requests
`-o` *file*	Set name of output file
`-O`	Generate optimized code (equivalent to `-xO2`)
`-P`	Run source thru preprocessor, output to `.i`
`-p`	Collect data for `prof`
`-Q[y|n]`	Add or don't add version stamp info
`-q[l|p]`	Collect data for `lprof` or `prof`
`-R`*dir*`[:`*dir*`]`	Specify library search path for dynamic linker
`-S`	Product `.s` file only (do not assemble or link)
`-s`	strip (4.1); pass to `ld` (5.0)
`-U`*name*	Undefine preprocessor symbol *name* as if by `#undef`
`-V`	Report versions of invoked programs
`-v`	Do stricter, `lint`-like semantic checking
`-W`*tool*`,`*arg(s)*	Hand off arguments to other components
`-w`	Do not print warnings
`-X[a,c,s,t]`	Compatibility options (ANSI, conformant, K&R C, transition)
`-xa`	Collect data for basic block profiling (`tcov`)
`-xF`	Produce code that can be re-ordered at function level
`-xlibmil`	Include inline templates as part of `-fast`
`-xlicinfo`	Return status of licensing system
`-xM`	Preprocess, send makefile dependencies to standard output
`-xnolibmil`	Reset `-fast` so that it does not include inline templates
`-xO[1,2,3,4]`	Generate optimized code (default is `-X02`)
`-xpg`	Collect data for `gprof`
`-xs`	places all stabs in `.stab` section
`-xsb`	Collect info for code browser

Table 4-1 Summary of `cc` Options (Sheet 3 of 3)

Option or Flag	Description
`-xsbfast`	Collect info for code browser, but do not compile
`-xstrconst`	Place string literals into read-only data segment
`-Y`*item*`,` *dir*	Change pathname for components

4.3 *Commonly Used* `cc` *Command Line Options*

Searching for a Header File

Recall that the first line of our sample program was

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

The format of that directive is the one you should use to include any of the standard header files that are supplied with the C compilation system. The angle brackets (<>) tell the preprocessor to search for the header file in the standard place for header files on your system, usually the `/usr/include` directory.

The format is different for header files that you have stored in your own directories:

```
#include "header.h"
```

The quotation marks (`" "`) tell the preprocessor to search for `header.h` first in the directory of the file containing the `#include` line, which will usually be your current directory, then in the standard place.

If your header file is not in the current directory, specify the path of the directory in which it is stored with the `-I` option to `cc`. Suppose, for instance, that you have included both `stdio.h` and `header.h` in the source file `mycode.c`:

```
#include <stdio.h>
#include "header.h"
```

Suppose further that `header.h` is stored in the directory `../defs`. The command

```
$ cc -I../defs mycode.c
```

will direct the preprocessor to search for `header.h` first in the current directory, then in the directory `../defs`, and finally in the standard place. It will also direct the preprocessor to search for `stdio.h` first in `../defs`, then

in the standard place — the difference being that the current directory is searched only for header files whose name you have enclosed in quotation marks.

You can specify the `-I` option more than once on the `cc` command line. The preprocessor will search the specified directories in the order they appear on the command line. Needless to say, you can specify multiple options to `cc` on the same command line:

```
$ cc -o prog -I../defs mycode.c
```

Preparing Your Program for Symbolic Debugging

When you specify the `-g` option to `cc`,

```
$ cc -g mycode.c
```

you arrange for the compiler to generate information about program variables and statements that will be used by the symbolic debugger `dbx`. The information supplied to `dbx` will allow you to use the symbolic debugger to trace function calls, display the values of variables, set breakpoints, and so on.

Preparing Your Program for Profiling

The various profilers for optimizing your source code are described briefly in Chapter 1, "Introduction to ANSI C," and extensively in *Profiling Tools*.

To use the profilers that are supplied with the C compilation system, you must do two things:

1. **Compile and link your program with a profiling option:**

```
$ cc -xpg -o prog mycode.c      (for gprof)
$ cc -ql -o prog mycode.c       (for lprof)
$ cc -xa -o prog mycode.c       (for tcov)
$ cc -qp -o prog mycode.c       (for prof; -p may replace -qp)
```

2. **Run the profiled program:**

```
$ prog
```

At the end of execution, data about your program's run-time behavior is written to a file in your current directory:

```
$ ls
gmon.out                        (for gprof)
mon.out                         (for prof)
mycode.c
mycode.d                        (for tcov)
prog
prog.cnt                        (for lprof:)
```

3. **Run the profiler:**

```
$ gprof prog > output.file
$ lprof -o prog > output.file
$ tcov mycode.c                 (produces mycode.tcov file)
$ prof prog > output.file
```

The files are inputs to the profilers.

See the *Profiling Tools* manual for more information on `gprof`(1), `lprof`(1), `tcov`(1), and `prof`(1).

Non-Standard Floating Point

IEEE 754 floating-point default arithmetic is "nonstop" and underflows are "gradual." What do we mean by these terms?

This explanation is by necessity rather over-simplified. See the *Numerical Computation Guide* for more rigorous descriptions.

Nonstop means that execution doesn't halt on things like division by zero, floating-point overflow, or invalid operation exceptions. For example, consider the following, where `x` is zero and `y` is positive:

```
z = y / x;
```

By default, `x` gets set to the value `+Inf`, and execution continues. With the `-fnonstd` option, however, this code causes an ungraceful exit (say, a core dump).

Here's how gradual underflow works. Suppose you have the following code:

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
     x = x / 10;
```

The first time through the loop, `x` is set to 1; the second time through, to 0.1; the third time through, to 0.01; and so on. Eventually, `x` will reach the lower limit of the machine's capacity to represent its value. What happens the next time the loop runs?

Let's say that the smallest number characterizable is

```
1.234567e-38
```

The next time the loop runs, the number is modified by "stealing" from the mantissa and "giving" to the exponent:

```
1.23456e-39
```

and, subsequently,

```
1.2345e-40
```

and so on. This is know as "gradual underflow" and it's the default behavior. In non-standard behavior, none of this "stealing" goes on; typically, `x` is simply set to zero.

The Parts of C

5.1 Introduction

This chapter is a guide to the ANSI C language (not K&R C) compiler. The level of presentation assumes some experience with C and familiarity with fundamental programming concepts.

The compilers are compatible with the C language described in the American National Standards Institute (ANSI) "American National Standard for Information Systems—Programming Language —C," document number ANSI X3.159-1989.

The standard language is referred to as *ANSI C* in this document. The notation *K&R C* refers to non-ANSI (or pre-ANSI) C..

Compilation Modes

The compilation system has the following compilation modes, which correspond to degrees of compliance with ANSI C. `-Xt` is the default mode:

`-Xa`

(*a = ANSI*) ANSI C plus K&R C compatibility extensions, *with* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the ANSI C intrepretation.

`-Xc`

(*c = conformance*) Maximally conformant ANSI C, without K&R C compatibility extensions. The compiler will reject programs that use non-ANSI C constructs.

`-Xs`

(*s = senescent*) The compiled language includes all features compatible with (pre-ANSI) K&R C. The computer warns about all language constructs that have differing behavior between ANSI C and the old K&R C.

`-Xt`

(*t = transition*) ANSI C plus K&R C compatibility extensions, *without* semantic changes required by ANSI C. Where K&R C and ANSI C specify different semantics for the same construct, the compiler will issue warnings about the conflict and use the K&R C interpretation. This is the default compiler mode.

Global Behavior: Value vs. Unsigned Preserving

A program that depends on unsigned-preserving arithmetic conversions will behave differently. This is considered to be the most serious change made by ANSI C to a widespread current practice.

In the first edition of Kernighan and Ritchie, *The C Programming Language* (Prentice-Hall, 1978), `unsigned` specified exactly one type; there were no `unsigned chars`, `unsigned shorts`, or `unsigned longs`, but most C compilers added these very soon thereafter.

In previous C compilers, the *unsigned preserving* rule is used for promotions: when an unsigned type needs to be widened, it is widened to an unsigned type; when an unsigned type mixes with a signed type, the result is an unsigned type.

The other rule, specified by ANSI C, came to be called *value preserving*, in which the result type depends on the relative sizes of the operand types. When an `unsigned char` or `unsigned short` is *widened*, the result type is `int` if an `int` is large enough to represent all the values of the smaller type. Otherwise the result type is `unsigned int`. The *value preserving* rule produces the *least surprise* arithmetic result for most expressions.

Only in the `-Xt` amd `-Xs` modes does the compiler use the *unsigned* preserving promotions; in the other modes, `-Xc` and `-Xa`, the *value* preserving promotion rules are used. No matter what the current mode may be, the compiler warns about each expression whose behavior might depend on the promotion rules used.

This warning is not optional because this is a serious change in behavior.

How To Use This Chapter

You can use this chapter either as a quick reference guide, or as a comprehensive summary of the language as implemented by the compilation system. Many topics are grouped according to their place in the ANSI-specified phases of translation, which describe the steps by which a source file is translated into an executable program.

Phases of Translation

The compiler processes a source file into an executable in eight conceptual steps, which are called *phases of translation*. While some of these phases may in actuality be folded together, the compiler behaves as if they occur separately, in sequence.

1. Trigraph sequences are replaced by their single-character equivalents. (*Trigraph sequences* are explained in "Trigraph Sequences" on page 78).
2. Any source lines that end with a backslash and new-line are spliced together with the next line by deleting the backslash and new-line.
3. The source file is partitioned into preprocessing tokens and sequences of white-space characters. Each comment is, in effect, replaced by one space character. (*Preprocessing tokens* are explained in "Preprocessing Tokens" on page 78).
4. Preprocessing directives are executed, and macros are expanded. Any files named in `#include` statements are processed from phase 1 through phase 4, recursively.
5. Escape sequences in character constants and string literals are converted to their character equivalents.

6. Adjacent character string literals and wide character string literals are concatenated.

7. Each preprocessing token is converted into a token. The resulting tokens are syntactically and semantically analyzed and translated. (*Tokens* are explained under "Tokens" on page 72).

8. All external object and function references are resolved. Libraries are linked to satisfy external references not defined in the current translation unit. All translator output is collected into a program image which contains information needed for execution.

Output from certain phases may be saved and examined by specifying option flags on the `compiler` command line.

The preprocessing token sequence resulting from Phase 4 can be saved by using the following options:

- `-P` leaves preprocessed output in a file with a `.i` extension.
- `-E` sends preprocessed output to the standard output.

Use the `-c` option to `cc` (or `acc`) to save output from Phase 7 in a file with a `.o` extension. The output of Phase 8 is the compilation system's final output: `( a.out)`.

5.2 *Source Files and Tokenization*

Tokens

A token is a series of contiguous characters that the compiler treats as a unit. Translation phase 3 partitions a source file into a sequence of tokens. Tokens fall into seven classes:

- Identifiers
- Keywords
- Numeric Constants
- Character Constants
- String literals
- Operators

- Other separators and punctuators

Identifiers

- Identifiers name things such as variables, functions, data types, and macros.
- Identifiers are made up of a combination of letters, digits, or underscore (_) characters.
- First character may not be a digit.

Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

Table 5-1 Identifiers

`asm`	`default`	`for`	`short`	`union`
`auto`	`do`	`goto`	`signed`	`unsigned`
`break`	`double`	`if`	`sizeof`	`void`
`case`	`else`	`int`	`static`	`volatile`
`char`	`enum`	`long`	`struct`	`while`
`const`	`extern`	`register`	`switch`	
`continue`	`float`	`return`	`typedef`	

The keyword `asm` is reserved in all compilation modes except `-Xc`. The keyword __asm is a synonym for `asm` and is available under all compilation modes, although a warning will be issued when it is used under the `-Xc` mode.

Constants

Integral Constants

- Decimal
  - Digits `0-9`.
  - First digit may not be `0` (zero).

- Octal
  - Digits `0-7`.
  - First digit must be `0` (zero).
- Hexadecimal
  - Digits `0-9` plus letters `a-f` or `A-F`. Letters correspond to decimal values 10-15.
  - Prefixed by `0x` or `0X` (digit zero).

Note – An octal or hexadecimal constant with the sign bit on is treated as an unsigned value. For example, 0x80 through 0xff will not fit into a char, 0x8000 through 0xffff will not fit into a short, and 0x80000000 through 0xffffffff will not fit into an int. If these values are used as initializers, the following error message results: warning: initializer does not fit.

- Suffixes
  - All of the above can be suffixed to indicate type, as follows:

Table 5-2 Data Type Suffixes

Suffix	Type
`u` *or* `U`	unsigned
`l` *or* `L`	long
`ll` *or* `LL`	long long[a]
`lu` *or* `LU` *or* `Lu` *or* `lU`	unsigned long
`llu` *or* `LLU` *or* `LLu` *or* `llU`	unsigned long longa

a. long long is not available in -Xc mode.

When assigning types to constants, the compiler uses the first of this list in which the value can be represented:

```
int
long int
unsigned long int
long long int (not available in -Xc mode)
unsigned long long int (not available in -Xc mode)
```

Floating Point Constants

Floating-point constants consist of integer part, decimal point, fraction part, an `e` or `E`, an optionally signed integer exponent, and a type suffix, one of `f`, `F`, `l`, or `L`. Each of these elements is optional; however one of the following must be present for the constant to be a floating point constant:

- A decimal point (preceded or followed by a number).
- An `e` with an exponent.
- Any combination of the above. Examples:

  xxx e *exp*
  xxx.
  .xxx

- Type determined by suffix; `f` or `F` indicates `float`, `l` or `L` indicates `long double`; otherwise type is `double`.

Character Constants

- One or more characters enclosed in single quotes, as in `'x'`.
- All character constants have type `int`.
- Value of a character constant is the numeric value of the character in the ASCII character set.
- A multiple-character constant that is not an escape sequence (see below) has a value derived from the numeric values of each character. For example, the constant '123' has a value of

Table 5-3 Multiple-character Constant (ASCII)

0	'3'	'2'	'1'

or 0x333231. In other, non-ANSI versions of C the value is

Table 5-4 Multiple-character Constant (non-ASCII)

0	'1'	'2'	'3'

or 0x313233.

- Character constants may not contain the character ' or new-line. To represent these characters, and some others that may not be contained in the source character set, the compiler provides the following escape sequences:

Table 5-5 Character Constants

Character	Abbreviation	Escape Sequence	Character	Abbreviation	Escape Sequence
new-line	NL (LF)	\n	audible alert	BEL	\a
horizontal tab	HT	\t	question mark	?	\?
vertical tab	VT	\v	double quote	"	\"
backspace	BS	\b	octal escape	*ooo*	*ooo*
carriage return	CR	\r	hexadecimal escape	*hh*	\x*hh*
formfeed	FF	\f	backslash	\	\\
single quote	'	\'			

If the character following a backslash is not one of those specified, the compiler will issue a warning and treat the backslash-character sequence as the character itself. Thus, `\q` will be treated as `q`. However, if you represent a character this way, you run the risk that the character may be made into an escape sequence in the future, with unpredictable results. An explicit new-line character is invalid in a character constant and will cause an error message.

- The octal escape consists of one to three octal digits.
- The hexadecimal escape consists of one or more hexadecimal digits.

Wide Characters and Multibyte Characters

- A wide character constant is a character constant prefixed by the letter `L`.
- A wide character has an external encoding as a multibyte character and an internal representation as the integral type `wchar_t`, defined in `stddef.h`.
- A wide character constant has the integral value for the multibyte character between single quote characters, as defined by the locale-dependent mapping function `mbtowc`.

String Literals

- One or more characters surrounded by double quotes, as in `"xyz"`.
- Initialized with the characters contained in the double quotes.
- Have `static` storage duration and type *array of characters*.
- The escape sequences described in "Character Constants" may also be used in string literals. A double quote within the string must be escaped with a backslash. New-line characters are not valid within a string.
- Adjacent string literals are concatenated into a single string. A null character, `\0`, is appended to the result of the concatenation, if any.
- String literals are also known as *string constants*.

Wide String Literals

- A wide-character string literal is a string literal immediately prefixed by the letter `L`.
- Wide-character string literals have type *array of* `wchar_t`.
- Wide string literals may contain escape sequences, and they may be concatenated like ordinary string literals.

Comments

Comments begin with the characters `/*` and end with the next `*/`.

```
/* this is a comment */
```

Comments do *not* nest.

If a comment appears to begin within a string literal or character constant, it will be taken as part of the literal or constant, as specified by the phases of translation.

```
char *p = "/* this is not a comment */";  /* but this is */
```

5.3 Preprocessing

- Preprocessing handles macro substitution, conditional compilation, and file inclusion.
- Lines beginning with # indicate a preprocessing control line. Spaces and tabs may appear before and after the #.
- Lines that end with a backslash character \ and new-line are joined with the next line by deleting the backslash and the new-line characters. This occurs (in translation phase 2) before input is divided into tokens.
- Each preprocessing control line must appear on a line by itself.

Trigraph Sequences

Trigraph sequences are three-character sequences that are replaced by a corresponding single character in Translation Phase 1. The trigraph sequences are provided as a way to specify characters that are not available on some terminals, but that the C language uses, as follows:

Table 5-6 Trigraph Sequences

Sequence	Replaced By	Sequence	Replaced By	Sequence	Replaced By
??=	#	??(	[	??<	{
??/	\	??)	]	??>	}
??'	^	??!	|	??-	

No other such sequences are recognized.

Preprocessing Tokens

A token is the basic lexical unit of the language. All source input must be formed into valid tokens by translation phase seven. Preprocessing tokens (pp-tokens) are a superset of regular tokens. Preprocessing tokens allow the source file to contain non-token character sequences that constitute valid preprocessing tokens during translation. There are four categories of preprocessing tokens:

- Header file names, meant to be taken as a single token.
- Preprocessing numbers (discussed in "Preprocessing Numbers" on page 79).
- All other single characters that are not otherwise (regular) tokens. See the example under "Preprocessing Numbers" on page 79.
- Identifiers, numeric constants, character constants, string literals, operators, and punctuators.

Preprocessing Numbers

- A preprocessing number is made up of a digit, optionally preceded by a period, and may be followed by letters, underscores, digits, periods, and any one of `e+ e- E+ E-`.
- Preprocessing numbers include all valid number tokens, plus some that are not valid number tokens. For example, in the macro definition:

```
#define R 2e ## 3
```

the preprocessing number `2e` is not a valid number. However, the preprocessing operator `##` will *paste* it together with the preprocessing number `3` when `R` is replaced, resulting in the preprocessing number `2e3`, which is a valid number. See "Preprocessing Operators" on page 79 for a discussion of the `##` operator.

Preprocessing Directives

Preprocessing Operators

The preprocessing operators are evaluated left to right, without any defined precedence.

#

A macro parameter preceded by the # preprocessing operator has its corresponding unexpanded argument tokens converted into a string literal. (Any double quotes and backslashes contained in character constants or part of string literals are escaped by a backslash). The # character is sometimes referred to as the *stringizing* operator. This rule applies only within function-like macros.

##

If a replacement token sequence (see "Macro Definition and Expansion" below) contains a ## operator, the ## and any surrounding white space are deleted and adjacent tokens are concatenated, creating a new token. This occurs only when the macro is expanded.

Macro Definition and Expansion

- An object-like macro is defined with a line of the form

```
#define identifier token-sequence_opt
```

where *identifier* will be replaced with *token-sequence* wherever *identifier* appears in regular text.

- A function-like macro is defined with a line of the form

```
#define identifier ( identifier-list_opt ) token-sequence_opt
```

where the macro parameters are contained in the comma-separated identifier-list. The token-sequence following the identifier list determines the behavior of the macro, and is referred to as the *replacement list*. There can be no space between the identifier and the (character. For example:

```
#define FLM(a,b) a+b
```

The replacement-list `a+b` determines that the two parameters `a` and `b` will be added.

- A function-like macro is invoked in normal text by using its identifier, followed by a (token, a list of token sequences separated by commas, and a) token. For example:

```
FLM(3,2)
```

- The arguments in the invocation (comma-separated token sequences) may be expanded, and they then replace the corresponding parameters in the replacement token sequence of the macro definition. Macro arguments in the invocation are *not* expanded if they are operands of # or ## operators in the replacement string. Otherwise, expansion does take place. For example:

Assume that `M1` is defined as `3`:

```
#define M1 3
```

When the function-like macro `FLM` is used, use of the `#` or `##` operators will affect expansion (and the result), as follows:

Table 5-7 Expansion of # and ## Macros

Definition	Invocation	Result	Expansion?
`a+b`	`FLM(M1,2)`	`3 + 2`	Yes, Yes
`#a`	`FLM(M1)`	`"M1"`	No
`a##b`	`FLM(M1,2)`	`M12`	No, No
`a+#b`	`FLM(M1,2)`	`3 + "2"`	Yes, No

In the last example line, the first `a` in `a+#a` is expanded, but the second `a` is not expanded because it is an operand of the `#` operator.

- The number of arguments in the invocation must match the number of parameters in the definition.
- A macro's definition, if any, can be eliminated with a line of the form:

```
#undef identifier
```

There is no effect if the definition doesn't exist.

File Inclusion

- A line of the form:

```
#include "filename"
```

causes the entire line to be replaced with the contents of *filename*. The following directories are searched, in order:

a. The current directory (of the file containing the `#include` line).

b. Any directories named in `-I` options to the compiler, in order.

c. A list of standard places, typically, but not necessarily, `/usr/include`.

- A line of the form:

```
#include <filename>
```

causes the entire line to be replaced with contents of *filename*. The angle brackets surrounding *filename* indicate that *filename* is not searched for in the current directory.

- A third form allows an arbitrary number of preprocessing tokens to follow the `#include`, as in:

```
#include preprocessing-tokens
```

The preprocessing tokens are processed the same way as when they are used in normal text. Any defined macro name is replaced with its replacement list of preprocessing tokens. The preprocessing tokens must expand to match one of the first two forms (< ... > or "...").

- A file name beginning with a slash / indicates the absolute pathname of a file to include, no matter which form of `#include` is used.
- Any `#include` statements found in an included file cause recursive processing of the named file(s).

Conditional Compilation

Different segments of a program may be compiled conditionally. Conditional compilation statements must observe the following sequence:

1. One of: `#if` or `#ifdef` or `#ifndef`.
2. Any number of optional `#elif` lines.
3. One optional `#else` line.
4. One `#endif` line.

- `#if` *integral-constant-expression*

  Is true, if *integral-constant-expression* evaluates to nonzero.

  If true, tokens following the `if` line are included.

The *integral-constant-expression* following the `if` is evaluated by following this sequence of steps:

1. Any preprocessing tokens in the expression are expanded. Any use of the `defined` operator evaluates to `1` or `0` if its operand is, respectively, defined, or not.
2. If any identifiers remain, they evaluate to `0`.
3. The remaining integral constant expression is evaluated. The constant expression must be made up of components that evaluate to an integral constant. In the context of a `#if`, the integral constant expression may not contain the `sizeof` operator, casts, or floating point constants.
   The following table shows how various types of constant expressions following a `#if` would be evaluated. Assume that *name* is not defined.

Table 5-8 Constant Expression Evaluation

Constant Expression	Step 1	Step 2	Step 3
`__STDC__`	1	1	1
`!defined(__STDC__)`	!1	!1	0
`3||`*name*	3 || *name*	3 || 0	1
`2 +` *name*	2 + *name*	2 + 0	2

- `#ifdef`
- *identifier*

  Is true if *identifier* is currently defined by `#define` or by the `-D` option to the `compiler` command line.

- `#ifndef` *identifier*

  Is true if *identifier* is not currently defined by `#define` (or has been undefined).

- `#elif` *constant-expression*

  Indicates alternate if-condition when all preceding if-conditions are false.

- `#else`

Indicates alternate action when no preceding `if` or `elif` conditions are true. A comment may follow the `else`, but a token may not.

- `#endif`

  Terminates the current conditional. A comment may follow the `endif`, but a token may not.

Line Control

- Useful for programs that generate C programs.
- A line of the form

```
#line constant "filename"
```

causes the compiler to believe, for the purposes of error diagnostics and debugging, that the line number of the next source line is equal to *constant* (which must be a decimal integer) and the current input file is *filename* (enclosed in double quotes). The quoted file name is optional. *constant* must be a decimal integer in the range 1 to `MAXINT`. `MAXINT` is defined in `limits.h`.

Assertions

A line of the form

```
#assert predicate (token-sequence)
```

associates the *token-sequence* with the predicate in the assertion name space (separate from the space used for macro definitions). The predicate must be an identifier token.

```
#assert predicate
```

asserts that *predicate* exists, but does not associate any token sequence with it.

The compiler provides the following predefined predicates by default:

```
#assert machine ( SPARC )
#assert system ( unix )
#assert cpu ( SPARC )
```

Any assertion may be removed by using `#unassert`, which uses the same syntax as `assert`. Using `#unassert` with no argument deletes all assertions on the predicate; specifying an assertion deletes only that assertion.

An assertion may be tested in a `#if` statement with the following syntax:

```
#if #predicate(non-empty token-list)
```

For example, the predefined predicate `system` can be tested with the following line:

```
#if #system(unix)
```

which will evaluate true.

Version Control

The `#ident` directive is used to help administer version control information.

```
#ident "version"
```

puts an arbitrary string in the `.comment` section of the object file. The `.comment` section is not loaded into memory when the program is executed.

Pragmas

Preprocessing lines of the form

```
#pragma pp-tokens
```

specify implementation-defined actions.

The following `#pragmas` are recognized by the compilation system:

- `#pragma fini` *identifier*
  (SunOS 5.0 only). Marking *identifier* as a "finalization function." Such functions are expected to be of type `void` and to accept no arguments, and are called either when a program terminates under program control or when the containing shared object is removed from memory. As with "initialization functions," finalization functions are executed in the order processed by the link editor(s).

- `#pragma init` *identifier*
  (SunOS 5.0 only). Marking *identifier* as an "initialization function." Such functions are expected to be of type `void` and to accept no arguments, and are called while constructing the memory image of the program at the start of execution. In the case of initializers in a shared object, they will be executed during the operation that brings the shared object into memory, either program start-up or some dynamic loading operation such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they were processed by the link editor(s), both static and dynamic.

- `#pragma ident` *string*
  Place *string* in the `.comment` section of the executable

- `#pragma int_to_unsigned` *function name*
  For a function that returns a type of `unsigned`, in `-Xt` or `-Xs` mode, change the function return to be of type `int`.

- `#pragma unknown_control_flow (`*name*`, [,`*name*`] )`
  Specifies a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`. Since such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

- #pragma unshared (*name* [, *name*])
  (SunOS 5.0 only). Any identifer named in the id list must be marked in the symbol table as unshared (thread-local), so that subsequent symbol table accesses for the symbol will be able to pass along this information to any tool that needs it. `errno` is an example of a symbol which should be marked.

- `#pragma weak` *function name* = *_function name*
  (SunOS 5.0 only). If a defined global symbol *function name* exists, the appearance of a weak symbol *_function name* with the same name will not cause an error.
- `#pragma weak` *function name*
  (SunOS 5.0 only). The linker will not complain if it does not find a definition for *function name*.

The compiler ignores unrecognized pragmas.

Error Generation

A preprocessing line consisting of

```
#error token-sequence
```

causes the compiler to produce a diagnostic message containing the *token-sequence*, and stop.

Predefined Names

The following identifiers are predefined as object-like macros:

Table 5-9 Pre-defined Identifiers

Identifier	Description
__LINE__	The current line number as a decimal constant.
__FILE__	A string literal representing the name of the file being compiled.
__DATE__	The date of compilation as a string literal in the form *"Mmm dd yyyy."*
__TIME__	The time of compilation, as a string literal in the form *"hh:mm:ss."*
__STDC__	The constant 1 under compilation mode -Xc, otherwise 0.

With the exception of __STDC__, these predefined names may not be undefined or redefined. Under compilation mode `-Xt`, __STDC__ may be undefined (`#undef __STDC__`) to cause a source file to think it is being compiled by a previous version of the compiler.

__STDC__ is not defined in -Xs mode.

5.4 Declarations and Definitions

Introduction

A declaration describes an identifier in terms of its type and storage duration. The location of a declaration (usually, relative to function blocks) implicitly determines the scope of the identifier.

Types

Basic Types

The basic types and their sizes are:

- char (1 byte)
- short int (2 bytes)
- int (4 bytes)
- long int (4 bytes)
- long long int (8 bytes)[1]

Each of char, short, int, long, and long long may be prefixed with signed or unsigned. A type specified with signed is the same as the type specified without signed.

- float (4 bytes)
- double (8 bytes)
- long double (16 bytes)
- void

Integral and floating types are collectively referred to as *arithmetic types*. Arithmetic types and pointer types (see "Pointer Declarators" on page 94) make up the *scalar types*.

1. long long is not available in -Xc mode.

Type Qualifiers

- `const`

  The compiler may place an object declared `const` in read-only memory. The program may not change its value and no further assignment may be made to it. An explicit attempt to assign to a `const` object will provoke an error.

- `volatile`

  `volatile` advises the compiler that unexpected, asynchronous events may affect the object so declared and warns it against making assumptions. An object declared `volatile` is protected from optimization that might otherwise occur.

Structures and Unions

- Structures

  A structure is a type that consists of a sequence of named members. The members of a structure may have different object types (as opposed to an array, whose members are all of the same type). To declare a structure is to declare a new type. A declaration of an object of type `struct` reserves enough storage space so that all of the member types can be stored simultaneously.

  A structure member may consist of a specified number of bits, called a bit-field. The number of bits (the size of the bit-field) is specified by appending a colon and the size (an integral constant expression, the number of bits) to the declarator that names the bit-field. The declarator name itself is optional; a colon and integer will declare the bit-field. A bit-field must have integral type. The size may be zero, in which case the declaration name must not be specified, and the next member starts on a boundary of the type specified. For example:

```
char :0
```

means "start the next member (if possible) on a `char` boundary." A named bit-field number that is not declared with an explicitly `unsigned` type holds values in the range

$$0 - (2^n-1)$$

where *n* is the number of bits. A bit-field declared with an explicit `signed` type holds values in the range

$$-2^{n-1}-(2^{n-1}-1)$$

An optional structure tag identifier may follow the keyword `struct`. The tag names the kind of structure described and `struct` may then be used as a shorthand name for the declarations that make up the body of the structure. For example:

```
struct t {
int x;
float y;
} st1, st2;
```

Here, `st1` and `st2` are structures, each made up of `x`, an `int`, and `y`, a `float`. The tag `t` may be used to declare more structures identical to `st1` and `st2`, as in:

```
struct t st3;
```

A structure may include a pointer to itself as a member; this is known as a *self-referential structure.*

```
struct n {
int x;
struct n *left;
struct n *right;
};
```

Note – Bit-fields of type `long long` are not permitted in structures or unions.

- Unions

  A union is an object that may contain one of several different possible member types. A union may have bit-field members. Like a structure, declaring a union declares a new type. Unlike a structure, a union stores the value of only one member at a given time. A union does, however, reserve enough storage to hold its largest member.

Enumerations

An enumeration is a unique type that consists of a set of constants called enumerators. The enumerators are declared as constants of type `int`, and optionally may be initialized by an integral constant expression separated from the identifier by an = character.

Enumerations consist of two parts:

- The set of constants.
- An optional tag.

For example:

```
enum color {red, blue=5, yellow};
```

`color` is the tag for this enumeration type. `red`, `blue`, and `yellow` are its enumeration constants. If the first enumeration constant in the set is not followed by an =, its value is `0`. Each subsequent enumeration constant not followed by an = is determined by adding 1 to the value of the previous enumeration constant. Thus `yellow` has the value 6.

```
enum color car_color;
```

declares `car_color` to be an object of type `enum color`.

Scope

The use of an identifier is limited to an area of program text known as the identifier's scope. The four kinds of scope are function, file, block, and function prototype.

- The scope of every identifier (other than label names) is determined by the placement of its declaration (in a declarator or type specifier).
- The scope of structure, union and enumeration tags begins just after the appearance of the tag in a type specifier that declares the tag. Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. Any other identifier has scope that begins just after the completion of its declarator.
- If the declarator or type specifier appears outside a function or parameter list, the identifier has file scope, which terminates at the end of the file (and all included files).
- If the declarator or type specifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which ends at the end of the block (at the `}` that closes that block).
- If the declarator or type specifier appears in the list of parameter declarations in a function prototype declaration, the identifier has function prototype scope, which ends at the end of the function declarator (at the `)` that ends the list).
- Label names always have function scope. A label name must be unique within a function.

Storage Class Specifiers

- `auto`

  An object may be declared `auto` only within a function. It has block scope and the defined object has automatic storage duration.

- `register`

  A `register` declaration is equivalent to an `auto` declaration. It also advises the compiler that the object will be accessed frequently.

- `static`

  `static` gives a declared object static storage duration (see "Storage Duration"). The object may be defined inside or outside functions. An identifier declared `static` with file scope has internal linkage. A function may be declared or defined with `static`. If a function is defined to be static, the function has internal linkage. A function may be declared with `static` at block scope; the function should be defined with `static` as well.

- `extern`

  `extern` gives a declared object static storage duration. An object or function declared with `extern` has the same linkage as any visible declaration of the identifier at file scope. If no file scope declaration is visible the identifier has external linkage.

- `typedef`

  Using `typedef` as a storage class specifier does not reserve storage. Instead, `typedef` defines an identifier that names a type.

Table 5-10 Storage Classes in C

Storage class	Declaration in C	Scope and initialization by compiler
Automatic	`auto int a;` `int a;`	Local to block or function in which they are declared. Values do not persist. Not initialized by compiler.
Register	`register int a;`	Local to block or function in which they are declared. Values do not persist. Not initialized by compiler.
Static	`static int a;`	Local to function in which they are declared. Values persist. Initialized to 0 at compile time.
External	`extern int a;`	Globally available to any function if declared outside and above that function. Globally available to all functions, regardless of number of source files, if declared as extern within each function. Values persist. Initialized to 0 at compile time.

Storage Duration

- Automatic Storage Duration

  Storage is reserved for an automatic object, and is available for the object on each entry (by any means) into the block in which the object is declared. On any kind of exit from the block, storage is no longer reserved.

- Static Storage Duration

An object declared outside any block, or declared with the keywords `static` or `extern`, has storage reserved for it for the duration of the entire program. The object retains its last-stored value throughout program execution.

Declarators

A brief summary of the syntax of declarators:

declarator:
    pointer$_{opt}$ *direct-declarator*
direct-declarator:
    identifier
    (*declarator*)
    direct-declarator [*constant-expression*$_{opt}$]
    direct-declarator (*parameter-type-list*)
    direct-declarator (*identifier-list*$_{opt}$)
pointer:
    * *type-qualifier-list*$_{opt}$
    * *type-qualifier-list*$_{opt}$ *pointer*

Pointer Declarators

- Pointer to a type:

```
char *p;
```

`p` is a pointer to type `char`. `p` contains the address of a `char` object.

Care should be taken when pointer declarations are qualified with `const`:

```
const int *pci;
```

declares a pointer to a `const`-qualified (*read-only*) `int`.

```
int *const cpi;
```

declares a constant pointer to `int` that is itself *read-only*.

- Pointer to a pointer:

```
char **t;
```

`t` points to a character pointer.

- Pointer to a function:

```
int (*f)( );
```

`f` is a pointer to a function that returns an `int`.

- Pointer to `void`:

```
void *
```

A pointer to `void` may be converted to or from a pointer to any object or incomplete type, without loss of information. This "generic pointer" behavior was previously carried out by `char *`; a pointer to `void` has the same representation and alignment requirements as a pointer to a character type.

Array Declarators

- One-dimensional array:

```
int ia[10];
```

`ia` is an array of 10 integers.

- Two-dimensional array:

```
char d[4][10];
```

`d` is an array of 4 arrays of 10 characters each.

- Array of pointers:

```
char *p[7];
```

`p` is an array of seven character pointers.

An array type of unknown size is known as an *incomplete type.*

Function Declarators

- A function declaration includes the return type of the function, the function identifier, and an optional list of parameters.
- Function prototype declarations include declarations of parameters in the parameter list.
- If the function takes no arguments, the keyword `void` may be substituted for the parameter list in a prototype.
- A parameter type list may end with an ellipsis ", . . . " to indicate that the function may take more arguments than the number described. The comma is necessary only if it is preceded by an argument.
- The parameter list may be omitted, which indicates that no parameter information is being provided.

Examples:

```
void srand(unsigned int seed);
```

The function `srand` returns nothing; it has a single parameter which is an `unsigned int`. The name `seed` goes out of scope at the `)` and as such serves solely as documentation.

```
int rand(void);
```

The function `rand` returns an `int`; it has no parameters.

```
int strcmp(const char *, const char *);
```

The function `strcmp` returns an `int`; it has two parameters, both of which are pointers to character strings that `strcmp` does not change.

```
void (*signal(int, void (*)(int)))(int);
```

The function `signal` returns a pointer to a function that itself returns nothing and has an `int` parameter; the function `signal` has two parameters, the first of which has type `int` and the second is a pointer to a function which returns `void` (this "second" function itself has one argument of type `int`).

```
int fprintf(FILE *stream, const char *format, ...);
```

The function `fprintf` returns an `int`; `FILE` is a typedef name declared in `stdio.h`; `format` is a `const` qualified character pointer; note the use of ellipsis (. . .) to indicate an unknown number of arguments.

Function Definitions

A *function definition* includes the body of the function after the declaration of the function. As with declarations, a function may be defined as a function prototype definition or defined in the old style. The function prototype style includes type declarations for each parameter in the parameter list. This example shows how `main` would be defined in each style:

Table 5-11 Function Definitions

Function Prototype Style	Old Style

`int`
`main(int argc, char *argv[])`
`{`
`. . .`
`. . .`
`. . .`
`}`
	`int`
	`main(argc, argv)`
	`int argc;`
	`char *argv[];`
	`{`
	`. . .`
	`}`

Some important rules that govern function definitions:

- An old style definition names its parameters in an identifier list, and their declarations appear between the function declarator and the "`{`" that begins the function body.
- Under the old style, if the type declaration for a parameter was absent, the type defaulted to `int`. In the new style, all parameters in the parameter list must be type-specified and named. The exception to this rule is the use of ellipsis, explained in "Function Declarators" on page 96.
- A function definition serves as a declaration.

- Incomplete types are not allowed in the parameter list or as the return type of a function definition. They are allowed in other function declarations.

5.5 Conversions and Expressions

Implicit Conversions

Characters and Integers

Any of the following may be used in an expression where an `int` or `unsigned int` may be used.

- `char`.
- `short int`.
- A `char`, `short`, or `int` bit-field.
- The signed or unsigned varieties of any of the above types.
- An object or bit-field that has enumeration type.

If an `int` can represent all values of the original type, the value is converted to an `int`; otherwise it is converted to an `unsigned int`. This process is called *integral promotion.*

Note – The promotion rules for ANSI C are different from previous versions of C. The compiler warns about expressions where this may lead to different behavior.

Compilation Mode Dependencies

- Under compilation modes `-Xs` and `-Xt`, `unsigned char` and `unsigned short` are promoted to `unsigned int` (unsigned preserving).
- Under compilation modes `-Xa` and `-Xc`, `unsigned char` and `unsigned short` are promoted to `int`.

Signed and Unsigned Integers

- When an integer is converted to another integral type, the value is unchanged if the value can be represented by the new type.

- If a negative signed integer is converted to an unsigned integer with greater size, the signed integer is first promoted to the signed integer corresponding to the unsigned integer.

Integral and Floating

When a floating type is converted to any integral type, any fractional part is discarded.

Float and Double

A `float` is promoted to `double` or `long double`, or a `double` is promoted to `long double` without a change in value.

The actual rounding behavior that is used when a floating point value is converted to a smaller floating point value depends on the rounding mode in effect at the time of execution. The default rounding mode is "round to nearest." Chapter 3, "acc Compiler Options for SunOS 4.x," briefly describes "gradual underflow" behavior; see the *Numerical Computation Guide* and the *IEEE Standard for Binary Floating-Point Arithmetic* (ANSI/IEEE Std 754-1985) for a more complete discussion of rounding modes.

Usual Arithmetic Conversions

Some binary operators convert the types of their operands in order to yield a common type, which is also the type of the result. These are called the *usual arithmetic conversions:*

- If either operand is type `long double`, the other operand is converted to `long double`.
- Otherwise, if either operand has type `double`, the other operand is converted to `double`.
- Otherwise, if either operand has type `float`, the other operand is converted to `float`.
- Otherwise, the integral promotions are performed on both operands. Then, these rules are applied:
  - If either operand has type `unsigned long long int`, then the other operator is converted to `unsigned long long int`.[1]
  - If either operand has type `long long int`, then the other operator is converted to `long long int`.

- If either operand has type `unsigned long int`, the other operand is converted to `unsigned long int`.
- Otherwise, if one operand has type `long int` and the other has type `unsigned int`, both operands are converted to `unsigned long int`.
- Otherwise, if either operand has type `long int`, the other operand is converted to `long int`.
- Otherwise if either operand has type `unsigned int`, the other operand is converted to `unsigned int`.
- Otherwise, both operands have type `int`.

Expressions

`lvalues` *and Objects*

An object is a region of storage that can be manipulated. An `lvalue` is an expression referring to an object. An obvious example of an `lvalue` expression is an identifier. There are operators that yield `lvalues`: for example, if `E` is an expression of pointer type, then `*E` is an `lvalue` expression referring to the object to which `E` points.

An `lvalue` is *modifiable* if:

- it does not have array type,
- it does not have an incomplete type,
- it does not have a const-qualified type, and, if it is a structure or union, it does not have any member (including, recursively, any member of all contained structures or unions) with a const-qualified type.

The name *lvalue* comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an `lvalue` expression.

Primary Expressions

- Identifiers, constants, string literals, and parenthesized expressions are primary expressions.

1. `long long` is not available in `-Xc` mode.

- An identifier is a primary expression, provided it has been declared as designating an object (which makes it an `lvalue`) or a function (which makes it a function designator).
- A constant is a primary expression; its type depends on its form and value.
- A string literal is a primary expression; it is an `lvalue`.
- A parenthesized expression is a primary expression. Its type and value are identical to those of the unparenthesized version. It is an `lvalue`, a function designator, or a void expression, according to the type of the unparenthesized expression.

Operators

For a summary of operator associativity and precedence, see Table 5-12 on page 109.

Unary Operators

Expressions with unary operators group right to left.

`*` *e*

Indirection operator. Returns the object or function pointed to by its operand. If the type of the expression is "pointer to . . .," the type of the result is " . . . ".

`&` *e*

Address operator. Returns a pointer to the object or function referred to by the operand. Operand must be an `lvalue` or function type, and not a bit-field or an object declared `register`. Where the operand has type "*type*," the result has type "pointer to *type*."

`-` *e*

Negation operator. The operand must have arithmetic type. Result is the negative of its operand. Integral promotion is performed on the operand, and the result has the promoted type. The negative of an unsigned quantity is computed by subtracting its value from 2v'-0.4'*n*v'0.4' where *n* is the number of bits in the result type.

`+ e`

Unary plus operator. The operand must have arithmetic type. Result is the value of its operand. Integral promotion is performed on the operand, and the result has the promoted type.

`! e`

Logical negation operator. The operand must have arithmetic or pointer type. Result is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is `int`.

`~ e`

The ~ operator yields the one's complement (all bits inverted) of its operand, which must have integral type. Integral promotion is performed on the operand, and the result has the promoted type.

`+ + e`

The object referred to by the `lvalue` operand of prefix + + is incremented. The value is the new value of the operand but is not an `lvalue`. The expression + +`x` is equivalent to `x += 1`. The type of the result is the type of the operand.

`- - e`

The modifiable `lvalue` operand of prefix - - is decremented analogously to the prefix + + operator.

`e + +`

When postfix + + is applied to a modifiable `lvalue`, the result is the value of the object referred to by the `lvalue`. After the result is noted, the object is incremented in the same manner as for the prefix + + operator. The type of the result is the same as the type of the `lvalue`.

`e - -`

When postfix - - is applied to an `lvalue`, the result is the value of the object referred to by the `lvalue`. After the result is noted, the object is decremented in the same manner as for the prefix - - operator. The type of the result is the same as the type of the `lvalue`.

`sizeof` *e*

The `sizeof` operator yields the size in bytes of its operand. When applied to an object with array type, the result is the total number of bytes in the array. (The size is determined from the declarations of the objects in the expression.) This expression is semantically an `unsigned` constant (of type

`size_t`, a `typedef`) and may be used anywhere a constant is required (except in a `#if` preprocessing directive line). One major use is in communication with routines like storage allocators and I/O systems.

`sizeof` (*type*)
The `sizeof` operator may also be applied to a parenthesized type name. In that case, it yields the size in bytes of an object of the indicated type.

Cast Operators - Explicit Conversions

`(`*type*`)` *e*
Placing a parenthesized type name before an expression converts the value of the expression to that type. Both the operand and *type* must be pointer type or an arithmetic type.

Multiplicative Operators

The multiplicative operators `*`, `/`, and `%` group left to right. The usual arithmetic conversions are performed, and that is the type of the result.

e `*` *e*
Multiplication operator. The `*` operator is commutative.

e `/` *e*
Division operator. When positive integers are divided, truncation is toward 0. If either operand is negative, the quotient is negative. Operands must be arithmetic types.

e `%` *e*
Remainder (or modulus) operator. Yields the remainder from the division of the first expression by the second. The operands must have integral type. The sign of the remainder is that of the first operand. It is always true that `(a/b)*b + a%b` is equal to `a` (if `a/b` is representable).

Additive Operators

The additive operators + and – group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

e + *e*

Result is the sum of the operands. A pointer to an object in an array and an integral value may be added. The latter is in all cases converted to an address offset by multiplying it by the size of the object to which the pointer points. The result is a pointer of the same type as the original pointer that points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The + operator is commutative.

The valid operand type combinations for the + operator are:

a + *a*
p + *i* or *i* + *p*

where *a* is an arithmetic type, *i* is an integral type, and *p* is a pointer.

e - *e*

Result is the difference of the operands. The operand combinations are the same as for the + operator, except that a pointer type may not be subtracted from an integral type.

Also, if two pointers to objects of the same type are subtracted, the result is converted (by division by the size of the object) to an integer that represents the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object size. The result type is `ptrdiff_t` (defined in `stddef.h`). `ptrdiff_t` is a typedef for `int` in this implementation. It should be used "as is" to ensure portability. Valid type combinations are

a - *a*
p - *i*
p - *p*

Bitwise Shift Operators

The bitwise shift operators << and >> take integral operands.

e1 << *e2*
Shifts *e1* left by *e2* bit positions. Vacated bits are filled with zeros.

e1 >> *e2*
Shifts *e1* right by *e2* bit positions.

The result types of the bitwise shift operators are compilation-mode dependent, as follows:

`-Xt`
The result type is `unsigned` if either operand is `unsigned`.

`-Xa, -Xc`
The result type is the promoted type of the left operand. Integral promotion occurs before the shift operation.

Relational Operators

```
a relop a
p relop p
```

- The relational operators < (less than) > (greater than) <= (less than or equal to) >= (greater than or equal to) yield `1` if the specified relation is true and `0` if it is false.
- The result has type `int`.
- Both operands:
  - have arithmetic type; or
  - are pointers to qualified or unqualified v the"`ions`of

Equality Operators

```
a eqop a
p eqop p
p eqop 0
0 eqop p
```

- The = = (equal to) and != (not equal to) operators are analogous to the relational operators; however, they have lower precedence.

Bitwise AND Operator

```
ie1 & ie2
```

- Bitwise "and" of *ie1* and *ie2*.
- Value contains a `1` in each bit position where both *ie1* and *ie2* contain a `1`, and a `0` in every other position.
- Operands must be integral; the usual arithmetic conversions are applied, and that is the type of the result.

Bitwise Exclusive OR Operator

```
ie1 ^ ie2
```

- Bitwise exclusive "or" of *ie1* and *ie2*.
- Value contains a `1` in each position where there is a `1` in either *ie1* or *ie2*, but not both, and a `0` in every other bit position.
- Operands must be integral; the usual arithmetic conversions are applied, and that is the type of the result.

Bitwise OR Operator

```
ie1 | ie2
```

- Bitwise inclusive "or" of *ie1* and *ie2*.
- Value contains a `1` in each bit position where there is a `1` in either *ie1* or *ie2*, and a `0` in every other bit position.
- Operands must be integral; the usual arithmetic conversions are applied, and that is the type of the result.

Logical AND Operator

```
e1 && e2
```

- Logical "and" of *e1* and *e2*.

- *e1* and *e2* must be scalars.
- *e1* is evaluated first, and *e2* is evaluated only if *e1* is nonzero.
- Result is `1` if both *e1* and *e2* are non-zero, otherwise `0`.
- Result type is `int`.

Logical OR Operator

```
e1 || e2
```

- Logical "or" of *e1* and *e2*.
- *e1* and *e2* must be scalars.
- *e1* is evaluated first, and *e2* is evaluated only if *e1* is zero. Result is 0 only if both *e1* and *e2* are false, otherwise 1.
- Result type is `int`.

Conditional Operator

```
e ? e1 : e2
```

- If *e* is nonzero, then *e1* is evaluated; otherwise *e2* is evaluated. The value is *e1* or *e2*.
- The first operand must have scalar type.
- For the second and third operands, one of the following must be true:
  - Both must be arithmetic types. The usual arithmetic conversions are performed to make them a common type and the result has that type.
  - Both must have compatible structure or union type; the result is that type.
  - Both operands have `void" type; the result has void` type.
  - Both operands are pointers to qualified or unqualified versions of compatible types. The result type is the composite type.
- One operand is a pointer and the other is a null pointer constant. The result type is the pointer type.

- One operand is a pointer to an object or incomplete type and the other is a pointer to a qualified or unqualified version of `void`. The result type is a pointer to `void`. For the pointer cases (the last three), the result is a pointer to a type qualified by all the qualifiers of the types pointed to by the operands.

Assignment Expressions

- Assignment operators are:

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

- An expression of the form *e1 op= e2* is equivalent to *e1 = e1 op (e2)* except that *e1* is evaluated only once.
- The left operand:
  - must be a modifiable `lvalue`.
  - must have arithmetic type, or, for `+=` and `-=`, must be a pointer to an object type and the right operand must have integral type.
  - of an `=` operator, if the operand is a structure or union, must not have any member or submember qualified with `const`.
- Result type is the type of the (unpromoted) left operand.

Comma Operator

e1 , *e2*

- *e1* is evaluated first, then *e2*. The result has the type and value of *e2* and is not an `lvalue`.

Structure Operators

`su.`*mem*

Indicates member *mem* of structure or union *su*.

`sup ->` *mem*

Indicates member *mem* of structure or union pointed to by *sup*. Equivalent to `(*sup).`*mem*.

Asociativity and Precedence of Operators

Table 5-12 Associativity and Precedence of Operators

		Operators	Associativity
Precedence	1	`( ) [ ] -> \.`	left to right
	2	`! ~ + + - - + - * & (type) sizeof`	right to left
	3	`* / %`	left to right
	4	`+ -`	left to right
	5	`<< >>`	left to right
	6	`< <= > >=`	left to right
	7	`== !=`	left to right
	8	`&`	left to right
	9	`^`	left to right
	10	`|`	left to right
	11	`&&`	left to right
	12	`||`	left to right
	13	`?:`	right to left
	14	`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
	15	`,`	left to right

Unary +, -, and * have higher precedence than their binary versions.

Prefix + + and - - have higher precedence than their postfix versions.

Constant Expressions

- A constant expression is evaluated during compilation (rather than at run time). As a result, a constant expression may be used any place that a constant is required.

- Constant expressions must not contain assignment, + +, - -, function-call, or comma operators, except when they appear within the operand of a `sizeof` operator.

Initialization

- Scalars (all arithmetic types and pointers):

  Scalar types with static or automatic storage duration are initialized with a single expression, optionally enclosed in braces. Example:

```
int i = 1;
```

  Additionally, scalar types (with automatic storage duration only) may be initialized with a nonconstant expression.

- Unions:

  An initializer for a union with static storage duration must be enclosed in braces, and initializes the first member in the declaration list of the union. The initializer must have a type that can be converted to the type of the first union member. Example:

```
union {
int i;
float f;
} u = {1}; /* initialize u.i */
```

  For a union with automatic storage duration, if the initializer is enclosed in braces, it must consist of constant expressions that initialize the first member of the union. If the initializer is not enclosed in braces, it must be an expression that has the matching union type.

- Structures:

The members of a structure may be initialized by initializers that can be converted to the type of the corresponding member.

```
struct s {
int i;
char c;
char *s;
} st = { 3, 'a', "abc" };
```

This example illustrates initialization of all three members of the structure. If initialization values are missing, as in

```
struct s st2 = {5};
```

then the first member is initialized (in this case, member `i` is initialized with a value of `5`), and any uninitialized member is initialized with `0` for arithmetic types and a null pointer constant for pointer types.

For a structure with automatic storage duration, if the initializer is enclosed in braces, it must consist of constant expressions that initialize the respective members of the structure. If the initializer is not enclosed in braces, it must be an expression that has the matching structure type.

- Arrays:

  The number of initializers for an array must not exceed the dimension, (i.e., the declared number of elements), but there may be fewer initializers than the number of elements. When the number of initializers is less than the size of the array, the first array elements are initialized with the values given, until the supply of initializers is exhausted. Any remaining array elements are initialized with the value 0 or a null pointer constant, as explained above in the discussion of structures. Example:

```
int ia[5] = { 1, 2 };
```

In this example, an array of five `int`s is declared, but only the first two members are initialized explicitly. The first member, `ia[0]`, is initialized with a value of `1`; the second member, `ia[1]`, is initialized with a value of `2`. The remaining members are initialized with a value of `0`.

When no dimensions are given, the array is sized to hold exactly the number of initializers supplied.

A character array may be initialized with a string literal, as in:

```
char ca[ ] = { "abc" }; /*curly braces are optional*/
```

where the size of the array is four (three characters with a null byte appended). The following:

```
char cb[3] = "abc";
```

is valid; however, in this case the null byte is discarded. But:

```
char cc[2] = "abc";
```

is erroneous because there are more initializers than the array can hold.

Arrays may be initialized similarly with wide characters:

```
wchar_t wc[ ] = L"abc";
```

Initializing subaggregates (for example, arrays of arrays) requires the proper placement of braces. For example,

```
int ia [4][2] =
{
     1,
     2,
     3,
     4
};
```

initializes the first two rows of `ia` (`ia[0][0]`, `ia[0][1]`, `ia[1][0]`, and `ia[1][1]`), and initializes the rest to `0`. This is a *minimally bracketed* initialization.

Note that a similar *fully bracketed* initialization yields a different result:

```
int ia [4][2] =
{
      {1},
      {2},
      {3},
      {4},
};
```

initializes the first column of `ia` (`ia[0][0]`, `ia[1][0]`, `ia[2][0]`, and `ia[3][0]`), and initializes the rest to `0`.

Mixing the fully and minimally bracketed styles may lead to unexpected results. Use one style or the other consistently.

5.6 Statements

Expression Statement

```
expression;
```

The *expression* is executed for its side effects, if any (such as assignment or function call).

Compound Statement

```
{
      declaration-list_opt
      statement-list_opt
}
```

- Delimited by `{` and `}`.
- May have a list of *declarations*.
- May have a list of *statements*.
- May be used wherever *statement* appears below.

Selection Statements

if

```
if (expression)
      statement
```

- If *expression* evaluates to nonzero (true), *statement* is executed.
- If *expression* evaluates to zero (false), control passes to the statement following *statement*.
- The *expression* must have scalar type.

else

```
if (expression1)
      statement1
else if (expression2)
      statement2
else
      statement3
```

- If *expression1* is true, *statement1* is executed, and control passes to the statement following *statement3*. Otherwise, *expression2* is evaluated.
- If *expression2* is true, *statement2* is executed, and control passes to the statement following *statement3*. Otherwise, *statement3* is executed, and control passes to the statement following *statement3*.
- An `else` is associated with the lexically nearest `if` that has no `else` and that is at the same block level.

switch

```
switch (expression)
      statement
```

- Control jumps to or past *statement* depending on the value of *expression*.
- *expression* must have integral type.

- Any optional `case` is labeled by an integral constant expression.
- If a `default` case is present, it is executed if no other case match is found.
- If no `case` matches, including `default`, control goes to the statement following *statement*.
- If the code associated with a `case` is executed, control falls through to the next `case` unless a `break` statement is included.
- Each case of a switch must have a unique constant value after conversion to the type of the controlling expression. In practice, *statement* is usually a compound statement with multiple cases, and possibly a `default`; the description above shows the minimum usage. In the following example, `flag` gets set to 1 if `i` is 1 or 3, and to 0 otherwise:

```
switch (i) {
case 1:
case 3:
     flag = 1;
     break;
default:
     flag = 0;
}
```

Iteration Statements

while

```
while (expression)
     statement
```

This sequence is followed repetitively:

- *expression* is evaluated.
- If *expression* is non-zero, *statement* is executed.
- If *expression* is zero, *statement* is not executed, and the repetition stops.
- *expression* must have scalar type.

do-while

```
do
      statement
while (expression);
```

This sequence is followed repetitively:

- *statement* is executed.
- *expression* is evaluated.
- If *expression* is zero, repetition stops.

(`do-while` tests loop at the bottom; `while` tests loop at the top.)

for

```
for (expression1; expression2; expression3)
      statement
```

- *expression1* initializes the loop.
- *expression2* is tested before each iteration.
- If *expression2* is true:
  - *statement* is executed.
  - *expression3* is evaluated.
  - Loop until *expression2* is false (zero).
- Any of *expression1*, *expression2*, or *expression3* may be omitted, but not the semicolons.
- *expression1* and *expression3* may have any type; *expression2* must have scalar type.

Jump Statements

goto

```
goto identifier;
```

- Goes unconditionally to statement labeled with *identifier.*
- Statement is labeled with an identifier followed by a colon, as in:

```
A2: x = 5;
```

- Useful to break out of nested control flow statements.
- Can only jump within the current function.

break

Terminates nearest enclosing `switch`, `while`, `do`, or `for` statement. Passes control to the statement following the terminated statement. Example:

```
for (i=0; i<n; i++) {
     if ((a[i] = b[i]) = = 0)
          break;      /* exit for */
}
```

continue

Goes to top of smallest enclosing `while`, `do`, or `for` statement, causing it to reevaluate the controlling expression. A `for` loop's *expression3* is evaluated before the controlling expression. Can be thought of as the opposite of the `break` statement. Example:

```
for (i=0; i<n; i++) {
     if (a[i] != 0)
          continue;
     a[i] = b[i];
     k++;
}
```

return

```
return;
return expression;
```

- `return` by itself exits a function.
- `return` *expression* exits a function and returns the value of *expression*. For example:

```
return a + b;
```

5.7 Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of variables declared with `register` that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified. For example, in the expression

```
a[i] = b[i++]
```

the value of `i` could be incremented after `b[i]` is fetched, but before `a[i]` is evaluated and assigned to, or it could be incremented after the assignment.

The value of a multi-character character constant may be different for different machines.

Fields are assigned to words, and characters to integers, right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally imposed storage layouts.

The `lint` tool is useful for finding program bugs and non-portable constructs. For information on how to use `lint`, see the *Profiling Tools* manual

C Error Messages

6.1 Introduction

This chapter contains the text and explanation for all the warning and error messages produced by the ANSI C compiler.

Note – The compiler will display many of the messages shown here only when used with the `cc -v` option or `acc -vc` option. With this option, the compiler performs stricter semantics checking and, therefore, displays more diagnostic messages.

`lint(1)` is a program that checks your C code for errors. In many cases, `lint` will warn you about incorrect, dangerous, or non-standard code that the compiler will not necessarily flag. We strongly recommend that you run `lint` on your source code before compiling.

Many warning and error messages are common to both `lint` and the compiler. The manual *Profiling Tools* contains a chapter on warning messages produced *only* by `lint` (i.e., those not also produced by the compiler).

The message entries are formatted as follows:

Table 6-1 Explanation of Compiler Diagnostics

Entry		Comment
`n extra byte(s) in string literal initializer ignored`		*Text of message.*
Type: Warning	*Options:* all	*Type of message and command-line options which must be set for the message to appear ("all" indicates that the message is independent of options).*
A string literal that initializes a character array contains n more characters than the array can hold.		*Explanation of message.*
`char ca[3] = "abcd";`		*Example of code that might generate the message.*

Messages in this chapter are listed in alphabetic order according to the first letter in the message. Symbols (but not numbers) are not used for ordering. Therefore a messages such as

```
#elif has no preceding #if
```

will be found under the letter *E*, as will the message

```
")" expected
```

When an error occurs, the error message is preceded by a file name and line number. The line number is usually the line on which a problem has been diagnosed. Occasionally the compiler must read the next token before it can diagnose a problem, in which case the line number in the message may be a higher line number than that of the offending line.

Note that `lint`(1) issues all of the messages listed in this chapter, and additional messages about potential bugs and portability problems.

Message Types and Applicable Options

Each message description includes a *Type* and an *Options* field as follows:

Type

indicates whether the message is a warning, an error, a fatal error, or a combination of error types (see below).

Options
indicates which `compiler` command options must be set for the message to appear. "all" implies that the message is independent of `compiler` options.

The following paragraphs explain the differences between warnings, errors, and fatals.

Warning messages, in which the word `warning:` appears after the file name and line number, provide useful information without interrupting compilation. They may diagnose a programming error, or a violation of C syntax or semantics, for which the compiler will nevertheless generate valid object code.

Error messages, which lack the `warning:` prefix, will cause the `compiler` command to fail. Errors occur when the compiler has diagnosed a serious problem that makes it unable to understand the program or to continue to generate correct object code. It will attempt to examine the rest of your program for other errors, however. The `compiler` will not link your program if the compiler diagnoses errors.

Fatal errors cause the compiler to stop immediately and return an error indication to the `compiler` command. A fatal error message is prefixed with the word `fatal:`. Such messages typically apply to start-up conditions, such as being unable to find a source file.

Operator Names in Messages

Some messages include the name of a compiler operator, as in:

```
operands must have arithmetic type:  op "+"
```

Usually the operator in the message is a familiar C operator. At other times the compiler uses its internal name for the operator, like `U-` . "Operator Names" on page 228 lists these internal names and describes what they mean.

6.2 Messages

`0 is invalid in # <number> directive`	
Type: Error	*Options:* All
The line number in a line number information directive (which the compiler uses for internal communication) must be a positive, non-zero value.	
`# 0 "foo.c"`	

`0 is invalid in #line directive`	
Type: Error	*Options:* All
This diagnostic is similar to the preceding one, except the invalid line number appeared in a `#line` directive.	
`#line 0`	

`a cast does not yield an lvalue`	
Type: Warning, Error	*Options:* All
You may not apply a cast to the operand that constitutes the object to be changed in an assignment operation. The diagnostic is a warning if the size of the operand type and the size of the type being cast to are the same; otherwise it is an error.	

```
f(void){
    int i;
    (long) i = 5;
    (short) i = 4;
}
```

`\a is ANSI C "alert" character`	
Type: Warning	*Options:* `-Xt`
In other (K&R) C compilers, '\a' was equivalent to 'a'. However, ANSI C defines '\a' to be an alert character. In this implementation, the corresponding character code is 07, the BEL character.	
`int c = '\a';`	

`access through "void" pointer ignored`	
Type: Warning	*Options:* All
A pointer to void may not be used to access an object. You wrote an expression that does an indirection through a (possibly qualified) pointer to void. The indirection is ignored, although the rest of the expression (if any) is honored.	

`f(void){`
`  volatile void *vp1, *vp2;`
`  *(vp1 = vp2);/* assignment does get done */`
`}`	

`ANSI C behavior differs; not modifying typedef with "`*modifier*`"`	
Type: Warning	*Options:* `-Xa -Xc`
A `typedef`ed type may not be modified with the `short`, `long`, `signed`, or `unsigned` type modifiers, although earlier versions of C compilers permitted it. *modifier* is ignored. A related message is `modifying typedef with "`*modifier*`"; only qualifiers allowed.`	
`typedef int INT;` `unsigned INT ui;`	

`ANSI C predefined macro cannot be redefined`	
Type: Warning	*Options:* All
The source code attempted to define or redefine a macro that is predefined by ANSI C. The predefined macro is unchanged.	
`#define __FILE__ "xyz.c"`	

`ANSI C predefined macro cannot be undefined`	
Type: Warning	*Options:* All
The source code contains an attempt to undefine a macro that is predefined by ANSI C.	
`#undef __FILE__`	

`ANSI C requires formal parameter before "..."`	
Type: Warning	*Options:* `-Xc -v`
The K&R C implementation allows you to define a function with a variable number of arguments and no fixed arguments. ANSI C requires at least one fixed argument.	
`f(...){}`	

ANSI C treats constant as unsigned: op "*operator*"	
Type: Warning	*Options:* All

The type promotion rules for ANSI C are slightly different from those of previous versions of K&R C. In the current release the default behavior is to duplicate the previous rules. In future releases the default will be to use ANSI C rules. You may obtain the ANSI C interpretation by using the `-Xa` option for the `compiler` command.
Previous K&R C type promotion rules were "unsigned-preserving." If one of the operands of an expression was of unsigned type, the operands were promoted to a common unsigned type before the operation was performed.
ANSI C uses "value-preserving" type promotion rules. An unsigned type is promoted to a signed type if all its values may be represented in the signed type.
ANSI C also has a different rule from previous K&R C versions for the type of an integral constant that implicitly sets the sign bit.
The different type promotion rules may lead to different program behavior for the operators that are affected by the unsigned-ness of their operands:
The division operators: `/, /=, %, %=`.
The right shift operators: `>>, >>=`.
The relational operators: `<, <=, >, >=`.
The warning message tells you that your program contains an expression in which the behavior of *operator* will change in the future. You can guarantee the behavior you want by inserting an explicit cast in the expression.

```
f(void){
    int i;
    /* constant was integer in K&R C, unsigned in ANSI C */
    i /= 0xf0000000;
}
```

argument cannot have unknown size: arg #*n*	
Type: Error	*Options:* All

An argument in a function call must have a completed type. You passed a struct, union, or enum object whose type is incomplete.

```
f(void){
    struct s *st;
    g(*st);
}
```

`argument does not match remembered type: arg #n`	
Type: Warning	*Options:* `-v`

At a function call, the compiler determined that the type of the argument passed to a function disagrees with other information it has about the function. That other information comes from two sources:

- An old-style (non-prototype) function definition, or
- A function prototype declaration that has gone out of scope, but whose type information is still remembered.

The argument in question is promoted according to the default argument promotion rules.

This diagnostic may be incorrect if the old-style function definition case applies and the function takes a variable number of arguments.

```
void f(i)
int i;
{ }

void g(void)
{
    f("erroneous");
}
```

`argument is incompatible with prototype: arg #n`	
Type: Error	*Options:* All

You called a function with an argument whose type cannot be converted to the type in the function prototype declaration for the function.

```
struct s {int x;} q;
f(void){
    int g(int,int);
    g(3,q);
}
```

`argument mismatch`	
Type: Warning	*Options:* All
The number of arguments passed to a macro was different from the number in the macro definition.	

```
#define twoarg(a,b) a+b
int i = twoarg(4);
```

`argument mismatch: n1 arg[s] passed, n2 expected`	
Type: Warning	*Options:* `-v`
At a function call, the compiler determined that the number of arguments passed to a function disagrees with other information it has about the function. That other information comes from two sources: • An old-style (non-prototype) function definition, or • A function prototype declaration that has gone out of scope, but whose type information is still remembered. This diagnostic may be incorrect if the old-style function definition case applies and the function takes a variable number of arguments.	

```
extern int out_of_scope();
int f()
{          /* function takes no args */
    extern int out_of_scope(int);
}

int g()
{
    f(1);     /* f takes no args */
    out_of_scope(); /* out_of_scope expects one arg */
}
```

`array too big`	
Type: Error	*Options:* All
An array declaration has a combination of dimensions such that the declared object is too big for the target machine.	
`int bigarray[1000][1000][1000];`	

`asm() argument must be normal string literal`	
Type: Error	*Options:* All
The argument to an old-style asm() must be a normal string literal, not a wide one.	
`asm(L"wide string literal not allowed");`	

`"#assert identifier (..." expected`	
Type: Error	*Options:* All
In a `#assert` directive, the token following the predicate was not the (that was expected.	
`#assert system unix`	

]

`"#assert identifier" expected`	
Type: Error	*Options:* All
In an `#assert` directive, the token following the directive was not the name of the predicate.	
`#assert 5`	

"#assert" missing ")"	
Type: Error	*Options:* All
In a `#assert` directive, the parenthesized form of the assertion lacked a closing).	
`#assert system(unix`	

assignment type mismatch	
Type: Warning, Error	*Options:* All
The operand types for an assignment operation are incompatible. The message is a warning when the types are pointer types that do not match. Otherwise the message is an error.	

```
struct s { int x; } st;
f(void){
    int i;
    char *cp;
    const char *ccp;
    i = st;
    cp = ccp;
}
```

auto/register inappropriate here	
Type: Error	*Options:* All
A declaration outside any function has storage class auto or register.	

```
auto int i;
f(void){
}
```

`automatic redeclares external:` *name*	
Type: Warning	*Options:* All
You have declared an automatic variable *name* in the same block and with the same name as another symbol that is extern. ANSI C prohibits such declarations, but previous versions of K&R C allowed them. For compatibility with previous versions, references to *name* in this block will be to the automatic.	

`f(void){`
`extern int i;`
`int i;`
`}`	

`bad file specification`	
Type: Error	*Options:* All
The file specifier in a `#include` directive was neither a string literal nor a well-formed header name.	
`#include stdio.h`	

`bad octal digit:` *'digit'*	
Type: Warning	*Options:* `-Xt`
An integer constant that began with 0 included the non-octal digit *digit*. An 8 is taken to have value 8, and a 9 is taken to have value 9, even though they are invalid.	
`int i = 08;`	

`bad token in #error directive:` *token*	
Type: Error	*Options:* All
The tokens in a `#error` directive must be valid C tokens. The source program contained the invalid token *token*.	
`#error "this is an invalid token`	

`bad use of "#" or "##" in macro #define`	
Type: Warning	*Options:* All
In a macro definition, a # or ## operator was followed by a # or ## operator.	
`#define bug(s) # # s` `#define bug2(s) # ## s`	

`base type is really "`*type tag*`":` *name*	
Type: Warning	*Options:* `-Xt`
A type was declared with a struct, union, or enum type specifier and with tag and then used with a different type specifier to declare *type* is the type specifier that you used for the original declaration. For compatibility with previous releases of K&R C, the compiler treats the two types as being the same. In ANSI C (with the `-Xa` or `-Xc` options), the types are different.	

`struct s { int x,y,z; };`
`f(void){`
`union s foo;`
`}`	

`bit-field size <= 0:` *name*	
Type: Error	*Options:* All
The declaration for bit-field *name* specifies a zero or negative number of bits.	
`struct s { int x:-3; };`	

`bit-field too big:` *name*	
Type: Error	*Options:* All
The declaration for bit-field *name* specifies more bits than will fit in an object of the declared type.	
`struct s { char c:20; };`	

`"break" outside loop or switch`	
Type: Error	*Options:* All
A function contains a break statement in an inappropriate place, namely outside any loop or switch statement.	

`f(void){`
`  break;`
`}`	

`cannot access member of non-struct/union object`	
Type: Error	*Options:* All
The structure or union member must be completely contained within the left operand of the . operator.	

```
f(void){
    struct s { int x; };
    char c;
    c.x = 1;
}
```

`cannot begin macro replacement with "##"`	
Type: Warning	*Options:* All
The ## operator is a binary infix operator and may not be the first token in the macro replacement list of a macro definition.	

```
#define mac(s) ## s
```

`cannot concatenate wide and regular string literals`	
Type: Warning, Error	*Options:* All
Regular string literals and string literals for wide characters may be concatenated only if they are both regular or both wide. The compiler issues a warning if a wide string literal is followed by a regular one (and both are treated as wide); it issues an error if a regular string literal is followed by a wide one.	

```
#include <stddef.h>
wchar_t wa[] = L"abc" "def";
char a[] = "abc" L"def";
```

`cannot declare array of functions or void`	
Type: Error	*Options:* All
You have attempted to declare an array of functions or an array of `void`.	
`int f[5]();`	

`cannot define "defined"`	
Type: Warning	*Options:* All
The predefined preprocessing operator `defined` may not be defined as a macro name.	
`#define defined xyz`	

`cannot dereference non-pointer type`	
Type: Error	*Options:* All
The operand of the * (pointer dereference) operator must have pointer type. This diagnostic is also issued for an array reference to a non-array.	

```
f(void){
    int i;
    *i = 4;
    i[4] = 5;
}
```

`cannot do pointer arithmetic on operand of unknown size`	
Type: Error	*Options:* All
An expression involves pointer arithmetic for pointers to objects whose size is unknown.	

`f(void){`
`  struct s *ps;`
`  g(ps+1);`
`}`	

`cannot end macro replacement with "#" or "##"`	
Type: Warning	*Options:* All
A # or ## operator may not be the last token in the macro replacement list of a macro definition.	
`#define mac1(s) abc ## s ##` `#define mac2(s) s #`	

`cannot find include file:` *filename*	
Type: Error	*Options:* All
The file *filename* specified in an `#include` directive could not be located in any of the directories along the search path.	
`#include "where_is_it.h"`	

`cannot have void object:` *name*	
Type: Error	*Options:* All
You may not declare an object of type `void`.	
`void v;`	

`cannot initialize "extern" declaration:` *name*	
Type: Error	*Options:* All
Within a function, the declaration of an object with `extern` storage class may not have an initializer.	

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

`cannot initialize function:` *name*	
Type: Error	*Options:* All
A name declared as a function may not have an initializer.	
`int f(void) = 3;`	

`cannot initialize parameter:` *name*	
Type: Error	*Options:* All
Old-style function parameter *name* may not have an initializer.	
`int f(i)` `int i = 4; {}`	

`cannot initialize typedef:` *name*	
Type: Error	*Options:* All
A `typedef` may not have an initializer.	
`typedef int INT = 1;`	

`cannot open` *file*: *explanation*	
Type: Fatal	*Options:* all
The compiler was unable to open an input or output file. Usually this means the file name argument passed to the cc command was incorrect. *explanation* describes why *file* could not be opened.	
`cc badname.c -c x.c`	

`cannot open include file (too many open files):` *filename*	
Type: Error	*Options:* All
The compiler could not open a new include file, because too many other include files are already open. Such a situation could arise if you have that includes that includes and so on. The compiler supports at least eight levels of "nesting," up to a maximum defined by the operating system. The most likely reason for the diagnostic is that at some point an include file includes a file that had already been included. For example, this could happen if includes which includes again.	

`cannot recover from previous errors`	
Type: Error	*Options:* All
Earlier errors in the compilation have confused the compiler, and it cannot continue to process your program. Please correct those errors and try again.	

`cannot return incomplete type`	
Type: Error	*Options:* All
When a function is called that returns a structure or union, the complete declaration for the structure or union must have been seen already. Otherwise this message results.	

```
f(void){
    struct s g();
    g();
}
```

`cannot take address of bit-field:` *name*	
Type: Error	*Options:* All
You cannot take the address of a bit-field member of a structure or union.	

```
f(void){
    struct s { int x:3, y:4; } st;
    int *ip = &st.y;
}
```

`cannot take address of register:` *name*	
Type: Warning, Error	*Options:* all
You attempted to take the address of *name*, which is an object that was declared with the register storage class. You are not permitted to do so, whether or not the compiler actually allocates the object to a register. The attempt to take an object's address may have been implicit, such as when an array is dereferenced. The diagnostic is an error if a register was allocated for the object and a warning otherwise.	

```
f(void){
    register int i;
    register int ia[5];
    int *ip = &i;
    ia[2] = 1;
}
```

`cannot take sizeof bit-field:` *name*	
Type: Warning	*Options:* All
The `sizeof` operator may not be applied to bit-fields.	
`struct s { int x:3; } st;` `int i = sizeof(st.x);`	

`cannot take sizeof function:` *name*	
Type: Error	*Options:* All
The `sizeof` operator may not be applied to functions.	
`int f(void);` `int i = sizeof(f);`	

`cannot take sizeof void`	
Type: Error	*Options:* All
The `sizeof` operator may not be applied to type `void`.	
`void v(void);` `int i = sizeof(v());`	

`cannot undefine "defined"`	
Type: Warning	*Options:* All
The predefined preprocessing operator `defined` may not be undefined.	
`#undef defined`	

`case label affected by conversion:` *value*	
Type: Warning	*Options:* `-v`
The *value* for the `case` label cannot be represented by the type of the controlling expression of a `switch` statement. If the type of the `case` expression and the type of the controlling expression have the same size, and the actual bit representation of the case expression is unchanged, but its interpretation is different. For example, the controlling expression may have type `int` and the case expression may have type `unsigned int`. In the diagnostic, *value* is represented as a hexadecimal value if the `case` expression is unsigned, decimal if it is signed. In the examplebelow, `0xffffffffu` is not representable as an `int`. When the case expression is converted to the type of the controlling expression its effective value is `-1`. That is, the case will be reached if `i` has the value `-1`, rather than `0xffffffff`.	

`f(void){`
`  int i;`
 	
`  switch( i ){`
`  case 0xffffffffu:`
`  ;`
`  }`
`}`	

`"case" outside switch`	
Type: Error	*Options:* All
A case statement occurred outside the scope of any switch statement.	

`f(void){`
`  case 4:  ;`
`}`	

`character constant too long`	
Type: Warning	*Options:* All
The character constant contains too many characters to fit in an integer. Only the first four characters of a regular character constant, and only the first character of a wide character constant, are used. (Character constants that are longer than one character are non-portable.)	
`int i = 'abcde';`	

`character escape does not fit in character`	
Type: Warning	*Options:* All
A hexadecimal or octal escape sequence in a character constant or string literal produces a value that is too big to fit in an unsigned `char`. The value is truncated to fit.	
`char *p = "\x1ff\400";`	

`character escape does not fit in wide character`	
Type: Warning	*Options:* All
This message diagnoses a condition similar to the previous one, except the character constant or string literal is prefixed by L to designate a wide character constant or string literal. The character escape is too large to fit in an object of type wchar_t and is truncated to fit.	

`comment does not concatenate tokens`	
Type: Warning	*Options:* `-Xa, -Xc`
In previous releases of K&R C, it was possible to "paste" two tokens together by juxtaposing them in a macro with a comment between them. This behavior was never defined or guaranteed. ANSI C provides a well-defined operator, ##, that serves the same purpose and should be used. This diagnostic warns that the old behavior is not being provided.	
`#define PASTE(a,b) a/*GLUE*/b` `int PASTE(prefix,suffix) = 1; /* does not create "prefixsuffix" */`	

`comment is replaced by "##"`	
Type: Warning	*Options:* `-Xt`
This message is closely related to comment does not concatenate tokens . The diagnostic tells you that the compiler is treating an apparent concatenation as if it were the ## operator. The source code should be updated to use the new operator.	
`#define PASTE(a,b) a/*GLUE*/b` `int PASTE(prefix,suffix) = 1; /* creates "prefixsuffix" */`	

`const object should have initializer:` *name*	
Type: Warning	*Options:* `-v`
`A const` object cannot be modified. If you do not supply an initial value, the object will have a value of zero, or for automatics its value will be indeterminate.	
`const int i;`	

"continue" outside loop	
Type: Error	*Options:* All
Your program contains a `continue` statement outside the scope of any loop.	

`f(void){`
`  continue;`
`}`	

controlling expressions must have scalar type	
Type: Error	*Options:* All
The expression for an `if`, `for`, `while`, or `do-while` must be an integral, floating-point, or pointer type.	

`f(void){`
`  struct s {int x;} st;`
`  while (st) {}`
`}`	

conversion of double to float is out of range	
Type: Warning, Error	*Options:* All
A `double` expression has too large a value to fit in a `float`. The diagnostic is a warning if the expression is in executable code and an error otherwise.	
`float f = 1e300 * 1e300;`	

`conversion of double to integral is out of range`	
Type: Warning, Error	*Options:* All
A `double` constant has too large a value to fit in an integral type. The diagnostic is a warning if the expression is in executable code and an error otherwise.	
`int i = 1e100;`	

`conversion of floating-point constant to` *type* out of range	
Type: Error	*Options:* All
A floating-point constant has too large a value to fit in type *type*.	
`float f = 1e300f;`	

`-D option argument not an identifier`	
Type: Error	*Options:* All
An identifier must follow the -D cc command line option.	
`cc -D3b2 -c x.c`	

`-D option argument not followed by "="`	
Type: Warning	*Options:* All
If any tokens follow an identifier in a `-D` command line option to the `cc` command, the first such token must be `=`.	
`cc -DTWO+2 -c x.c`	

`declaration hides parameter:` *name*	
Type: Warning	*Options:* All
You have declared an identifier *name* with the same name as one of the parameters of the function. References to *name* in this block will be to the new declaration.	

```
int f(int i,int INT){
    int i;
    typedef int INT;
}
```

`declaration introduces new type in ANSI C:` *type tag*	
Type: Warning	*Options:* `-Xt`
`struct`, *`union`*, or `enum` *tag* has been redeclared in an inner scope. In previous versions of K&R C, this tag was taken to refer to the previous declaration of tag. In ANSI C, the declaration introduces a new When the `-Xt` option is selected, the compiler reproduces the earlier behavior.	

```
struct s1 { int x; };
f(void){
    struct s1;
    struct s2 { struct s1 *ps1; }; /* s1 refers to line 1 */
    struct s1 { struct s2 *ps2; };
}
```

`"default" outside switch`	
Type: Error	*Options:* All
A default label appears outside the scope of a switch statement.	

```
f(void){
default:  ;
}
```

`#define requires macro name`	
Type: Error	*Options:* All
A #define directive must be followed by the name of the macro to be defined.	
`#define +3`	

`digit sequence expected after "#line"`	
Type: Error	*Options:* All
The compiler expected to find the digit sequence that comprises a line number after `#line`, but the token it found there is either an inappropriate token or a digit sequence whose value is zero.	
`#line 09a`	

`directive is an upward-compatible ANSI C extension`	
Type: Warning	*Options:* `-Xc`
This diagnostic is issued when the compiler sees a directive that it supports, but that is not part of the ANSI C standard, and `-Xc` has been selected.	
`#assert system( unix )`	

`directive not honored in macro argument list`	
Type: Warning, Error	*Options:* All
A directive has appeared between the ( )'s that delimit the arguments of a function-like macro invocation. The following directives are disallowed in such a context: `#ident`, `#include`, `#line`, `#undef`. The diagnostic is a warning if it appears within a false group of an if-group, and an error otherwise.	

```
#define flm(a) a+4
int i = flm(
#ifdef flm/* allowed */
    #undef flm/* disallowed:  error */
    4
#else/* allowed */
    #undef flm/* disallowed:  warn */
    6
#endif/* allowed */
);
```

`division by 0`	
Type: Warning, Error	*Options:* All
An expression contains a division by zero that was detected at compile-time. If the division is part of a #if or #elif directive, the result is taken to be zero. The diagnostic is a warning if the division is in executable code, an error otherwise.	

```
f(void) {
    int i = 1/0;
}
```

`dubious` *type* declaration; use tag only: *tag*	
Type: Warning	*Options:* All

You declared a new `struct`, `union`, or `enum` *type* with tag *tag* within a function prototype declaration or the parameter declaration list of an old-style function definition, and the declaration includes a declarator list for *type*. Calls to the function would always produce a type mismatch, because the tag declaration goes out of scope at the end of the function prototype declaration or definition, according to ANSI C's scope rules. You could never declare an object of that type outside the function. You should declare the `struct`, `union`, or `enum` ahead of the function prototype or function definition and then refer to it just by its tag.
 	
The example below *should appear as*:
 	
`struct s {int x;};`
`int f(struct s st)`
`{}`	

`int f(struct s {int x;} st)` `{}`	

`dubious escape: \q`	
Type: Warning	*Options:* All
Only certain characters may follow \ in string literals and character constants; *q* was not one of them. ANSI C ignores the \.	
`int i = '\q';`	

`dubious escape: \<`*hex value*`>`	
Type: Warning	*Options:* All
This message diagnoses the same condition as the preceding one, but the character that follows \ in the program is a non-printing character. The *hex~value* between the brackets in the diagnostic is the character's code, printed as a hexadecimal number.	

`dubious reference to` *type* typedef: *typedef*	
Type: Warning	*Options:* All
This message is similar to `dubious tag in function prototype`: *type tag*. A function prototype declaration refers to a *type* `union`, `struct`, or `enum` typedef with name *typedef*. Because the `struct`, `union`, or `enum` has been declared within a function, it could not be in scope when you define the function whose prototype is being declared. The prototype declaration and function definition thus could never match.	

`f(void){`
`  struct s { int x; };`
`  typedef struct s ST;`
`  extern int g(ST, struct s);`
`}`	

`dubious static function at block level`	
Type: Warning	*Options:* `-Xc`
You declared a function with storage class `static` at block scope. The ANSI C standard says that the behavior is undefined if you declare a function at block scope with an explicit storage class other than `extern`. Although K&R C allowed you to declare functions this way, other implementations might not, or they might attach a different meaning to such a declaration.	

`void`
`f(void){`
`  static void g(void);`
`}`	

`dubious tag declaration:` *type tag*	
Type: Warning	*Options:* All
You declared a new `struct`, `union`, or `enum` type with tag *tag* within a function prototype declaration or the parameter declaration list of an old-style function definition. Calls to the function would always produce a type mismatch, because the tag declaration goes out of scope at the end of the function declaration or definition, according to ANSI C's scope rules. You could never declare an object of that type outside the function.	
`int f(struct s *);`	

`dubious tag in function prototype:` *type tag*	
Type: Warning	*Options:* All
This message is similar to the previous one. A function prototype declaration refers to a `struct`, `union`, or `enum` *type* with tag *tag*. The *tag* has been declared within a function. Therefore it could not be in scope when you define the function whose prototype is being declared. The prototype declaration and function definition thus could never match.	

```
f(void){
    struct s {int x;};
    int g(struct s *);
}
```

`duplicate case in switch:` *value*	
Type: Error	*Options:* All
There are two `case` statements in the current `switch` statement that have the same constant value *value*.	

```
f(void){
    int i = 5;
    switch(i) {
    case 4:
    case 4:
        break;
    }
}
```

`duplicate "default" in switch`	
Type: Error	*Options:* All
There are two `default` labels in the current `switch` statement.	

```
f(void){
    int i = 5;

    switch(i) {
    default:
    default:
        break;
    }
}
```

`duplicate formal parameter:` *name*	
Type: Warning	*Options:* All
In a function-like macro definition, *name* was used more than once as a formal parameter.	
`#define add3(a,a,c) a + b + c`	

`duplicate member name:` *member*	
Type: Error	*Options:* All
A `struct` or `union` declaration uses the name *member* for more than one member.	

```
union u {
    int i;
    float i;
};
```

`#elif follows #else`	
Type: Warning	*Options:* All
A preprocessing if-section must be in the order #if, optional #elif's, followed by optional #else and #endif. The code contains a #elif after the #else directive.	

```
#if defined(ONE)
    int i = 1;
#elif defined(TWO)
    int i = 2;
#else
    int i = 3;
#elif defined(FOUR)
    int i = 4;
#endif
```

`#elif has no preceding #if`	
Type: Error	*Options:* All
An `#elif` directive must be part of a preprocessing `if`-section, which begins with a `#if` directive. The code in question lacked the `#if`.	

`#elif defined(TWO)`
`    int i = 2;`
`#endif`	

`#elif must be followed by a constant expression`	
Type: Error	*Options:* All
There was no expression following the `#elif` directive.	

`#if defined(ONE)`
`    int i = 1;`
`#elif`
`    int i = 4;`
`#endif`	

`#else has no preceding #if`	
Type: Error	*Options:* All
An `#else` directive was encountered that was not part of a preprocessing `if`-section.	

`#else`
`    int i =7;`
`#endif`	

`embedded NUL not permitted in asm()`	
Type: Error	*Options:* All
The string literal that appears in an old-style `asm()` contains an embedded `NUL` character (character code 0).	
`asm("this is an old-style asm with embedded NUL:  \0");`	

`empty #assert directive`	
Type: Error	*Options:* All
An `#assert` directive contained no predicate name to assert.	
`#assert`	

`empty constant expression after macro expansion`	
Type: Error	*Options:* All
An `#if` or `#elif` directive contained an expression that, after macro expansion, consisted of no tokens.	

```
#define EMPTY
#if EMPTY
    char *mesg = "EMPTY is non-empty";
#endif
```

`empty #define directive line`	
Type: Error	*Options:* All
A `#define` directive lacked both the name of the macro to define and any other tokens.	
`#define`	

`empty file name`	
Type: Error	*Options:* All
The file name in a `#include` directive is null.	
`#include <>`	

`empty header name`	
Type: Error	*Options:* All
This diagnostic is similar to the preceding one, but the null file name arises after macro substitution.	
`#define NULLNAME <>` `#include NULLNAME`	

`empty predicate argument`	
Type: Error	*Options:* All
The compiler expects to find tokens between the ( )'s that delimit a predicate's assertions in a `#unassert` directive. None were present.	
`#unassert machine()`	

`empty translation unit`	
Type: Warning	*Options:* All
The source file has no tokens in it after preprocessing is complete. The ANSI C standard requires the compiler to diagnose a file that has no tokens in it.	

`#ifdef COMPILE`
`    int token;`
`#endif`	

`empty #unassert directive`	
Type: Error	*Options:* All
An `#unassert` contained no predicate name to discard.	
`#unassert`	

`empty #undef directive, identifier expected`	
Type: Error	*Options:* All
An `#undef` directive lacked the name of a macro to "undefine."	
`#undef`	

`{}-enclosed initializer required`	
Type: Warning	*Options:* All
When you initialize an aggregate, you must enclose the initializer in `{ }`'s, except when you initialize a character array with a string literal or an automatic structure with an expression .	

`int ia[5] = 1;`
`f(void){`
`  struct s { int x,y; } st = 1;`
`}`	

`/* encountered inside a comment`	
Type: Warning	*Options:* `-v`
There is a `/*` inside a comment.	

`/*  This is comment`
`    that has another /* inside`
`    of the comment`
`*/`	

`end-of-loop code not reached`	
Type: Warning	*Options:* All
You have written a loop in such a way that the code at the end of the loop that the compiler generates to branch back to the beginning of the loop is not reachable and will never be executed.	

`f(void){`
`    int i = 1;`
`    while (i) {`
`        return 4;`
`    }`
`}`	

`enum constants have different types: op "operator"`	
Type: Warning	*Options:* `-v`
You have used relational operator to compare enumeration constants from two different enumeration types. This may indicate a programming error. Note also that the sense of the comparison is known at compile time, because the constants' values are known.	

`enum e1 { ec11, ec12 } ev1;`
`enum e2 { ec21, ec22 } ev2;`
`void v(void){`
`  if (ec11 > ec22)`
`  ;`
`}`	

`enum type mismatch: arg #n`	
Type: Warning	*Options:* `-v`
The program is passing an enumeration constant or object to a function for which a prototype declaration is in scope. The passed argument is of a different enumerated type from the one in the function prototype, which may indicate a programming error.	

`enum e1 { ec11 } ev1;`
`enum e2 { ec21 } ev2;`
`void ef(enum e1);`
 	
`void v(void){`
`  ef(ec21);`
`}`	

`enum type mismatch: op "operator"`	
Type: Warning	*Options:* `-v`
This message is like the previous one. One of the operands of *operator* is an enumeration object or constant, and the other is an enumeration object or constant from a different enumerated type.	

```
enum e1 { ec11, ec12 } ev1;
enum e2 { ec21, ec22 } ev2;
void v(void){
    if (ev1 > ec22)
        ;
}
```

`enumeration constant hides parameter:` *name*	
Type: Warning	*Options:* All
A declaration of an enumerated type within a function includes an enumeration constant with the same name as parameter *name*. The enumeration constant hides the parameter.	

```
int
f(int i){
    enum e { l, k, j, i };
}
```

`enumerator used in its own initializer:` *name*	
Type: Warning	*Options:* All
When setting the value of enumerator *name* in an enumeration type declaration, you have used *name* in the expression. ANSI C's scope rules take name in the expression to be whatever symbol was in scope at the time.	

```
int i;
f(void){
    enum e { i = i+1, j, k };/* uses global i in i+1 */
}
```

`enumerator value overflows INT_MAX (2147483647)`	
Type: Error	*Options:* All
The value for an enumeration constant overflowed the maximum integer value.	
`enum e { e1=2147483647, e2 };/* overflow for e2 */`	

`EOF in argument list of macro:` *name*	
Type: Error	*Options:* All
The compiler reached end-of-file while reading the arguments for an invocation of function-like macro	
`#define mac(a)` `mac( arg1`	

`EOF in character constant`	
Type: Error	*Options:* All
The compiler encountered end-of-file inside a character constant.	

`EOF in comment`	
Type: Warning	*Options:* All
The compiler encountered end-of-file while reading a comment.	

`EOF in string literal`	
Type: Error	*Options:* All
The compiler encountered end-of-file inside a string literal.	

`#error:` *tokens*	
Type: Error	*Options:* All
A `#error` directive was encountered in the source file. The other *tokens* in the directive are printed as part of the message.	

`#define ONE 2`
`#if ONE != 1`
`#error ONE != 1`
`#endif`	

`error writing output file`	
Type: Error	*Options:* All
An output error occurred while the compiler attempted to write its output file or a temporary file. The most likely problem is that a file system is out of space.	

`")" expected`	
Type: Error	*Options:* All
In an `#unassert` directive, the assertion of a predicate to be dropped must be enclosed in `(~)`.	
`#unassert system(unix`	

"(" expected after "# identifier"	
Type: Error	*Options:* All
When the `# operator` is used in a `#if` or `#elif` directive to select a predicate instead of a like-named macro, the predicate must be followed by a parenthesized list of tokens.	

`#assert system(unix)`
`#define system "unix"`
`#if #system`
`  char *systype = system;`
`#endif`	

"(" expected after first identifier	
Type: Error	*Options:* All
In an `#unassert` directive, the assertion of a predicate to be dropped must be enclosed in `(~)`.	
`#unassert system unix`	

extern and prior uses redeclared as static: *name*	
Type: Warning	*Options:* `-Xc, -v`
You declared *name* at file scope as an `extern`, then later declared the same object or function as `static`. ANSI C rules require that the first declaration of an object or function give its actual storage class. K&R C accepts the declaration and treats the object or function as if the first declaration had been static.	
`extern int i;` `static int i;`	

n `extra bytes(s) in string literal initializer ignored`	
Type: Warning	*Options:* All
A string literal that initializes a character array contains *n* more characters than the array can hold.	
`char ca[3] = "abcd";`	

`first operand must have scalar type: op "?:"`	
Type: Error	*Options:* All
The conditional expression in a `?:` expression must have scalar (integral, floating-point, or pointer) type.	

`struct s { int x; } st;`
`f(void){`
`  int i = st ? 3 : 4;`
`}`	

`floating-point constant calculation out of range: op "`*operator*`"`	
Type: Warning, Error	*Options:* All
The compiler detected an overflow at compile time when it attempted the operator operation between two floating-point operands. The diagnostic is a warning if the expression is in executable code and an error otherwise.	
`double d1 = 1e300 * 1e300;`	

`floating-point constant folding causes exception`	
Type: Error	*Options:* All
This message is like the previous one, except that the operation caused a floating-point exception that causes the compiler to exit.	

`formal parameter lacks name: param #n`	
Type: Error	*Options:* All
In a function prototype definition, you failed to provide a name for the parameter.	
`int f(int){` `}`	

`function cannot return function or array`	
Type: Error	*Options:* All
You declared a function whose return type would be a function or array, rather than, perhaps, a pointer to one of those.	
`int f(void)[];/* function returning array of ints */`	

`function designator is not of function type`	
Type: Error	*Options:* All
You used an expression in a function call as if it were the name of a function or a pointer to a function when it was not.	

`f(void){`
`  char *p;`
`  p();`
`}`	

`function expects to return value:` *name*	
Type: Warning	*Options:* `-v`
The current function was declared with a type, but you used a `return` statement with no return value expression.	

```
f(void){
    return;
}
```

`function has no return statement:` *name*	
Type: Warning	*Options*: `-v`
The function should include a `return` statement.	

```
#include <stdio.h>
main(void)
{
    (void) printf("Do the hippy-hippy shake.\n");
}
```

`function prototype parameters must have types`	
Type: Warning	*Options:* All
A function prototype declaration cannot contain an identifier list; it must declare types. The identifier list is ignored.	

```
int f(i);
```

`identifier expected after "#"`	
Type: Error	*Options:* All
The compiler expected to find an identifier, a predicate name, after a `#` in a conditional compilation directive, and none was there.	

`#if #system(unix) || #`
`  char *os = "sys";`
`#endif`	

`identifier expected after #undef`	
Type: Error	*Options:* All
A `#undef` must be followed by the name of the macro to be undefined. The token following the directive was not an identifier.	
`#undef 4`	

`identifier or "-" expected after -A`	
Type: Error	*Options:* All
The `cc` command line argument `-A` must be followed by the name of a predicate to assert, or by a `-`, to eliminate all predefined macros and predicates. The token following `-A` was neither of these.	
`cc -A3b2 -c x.c`	

`identifier or digit sequence expected after "#"`	
Type: Error	*Options:* All
An invalid token or non-decimal number follows the # that introduces a preprocessor directive line.	
`#0x12`	

`identifier redeclared:` *name*	
Type: Warning	*Options:* All

You declared the identifier *name* in a way that is inconsistent with a previous appearance of or you declared *name* twice in the same scope.
Previous releases of K&R C were forgiving of inconsistent redeclarations if the types were "nearly" the same (such as `int` and `long` on a Sun SPARCstation). ANSI C considers the types to be different.

```
int x;
long x;
int y;
double y;
```

Declarations of functions with and without argument information can often lead to confusing diagnostics. The following example illustrates.

```
int f(char);
int f();
```

According to ANSI C's type compatibility rules, a function declaration that lacks type information (i.e., one that is not a function prototype declaration) is compatible with a function prototype only when each parameter type is unchanged by the default argument promotion rules. In the example, `char` would be affected by the promotion rules (it would be promoted to `int`). Therefore the two declarations have incompatible types.

`identifier redeclared; ANSI C requires "static":` *name*	
Type: Warning	*Options:* All
You declared *name* twice at file scope. The first one used storage class `static`, but the second one specified no storage class. ANSI C's rules for storage classes require that all redeclarations of *name* after the first must specify `static`.	
`static int i;` `int i;`	

`identifier redefined:` *name*	
Type: Error	*Options:* All
You have defined *name* more than once. That is, you have declared an object more than once with an initializer, or you have defined a function more than once.	
`int i = 1;` `int i = 1;`	

`#if must be followed by a constant expression`	
Type: Warning	*Options:* All
No expression appeared after a #if directive.	

`#if`
`    int i = 4;`
`#endif`	

`#if on line n has no #endif`	
Type: Error	*Options:* All
The compiler reached end of file without finding the #endif that would end the preprocessing if-section that began with the *if* directive that was on line The *if* directive is one of #if, #ifdef, or #ifndef.	
`#ifdef NOENDIF` `    int i = 1;`	

`#if-less #endif`	
Type: Error	*Options:* All
An #endif directive was encountered that was not part of a preprocessing if-section.	
`    int i = 1;` `#endif`	

`#ifdef must be followed by an identifier`	
Type: Warning	*Options:* All
A #ifdef preprocessing directive must be followed by the name of the macro to check for being defined. The source code omitted the identifier. The #ifdef is treated as if it were false.	

`#ifdef`
`    int i = 1;`
`#endif`	

`#ifndef must be followed by an identifier`	
Type: Warning	*Options:* All
The `#ifndef` directive must be followed by the identifier that is to be tested for having been defined.	

`#ifndef`
`    int i = 5;`
`#endif`	

`ignoring malformed #pragma fini`	
Type: Warning	*Options:* All
The compiler encountered a `#pragma fini` directive that did not have the form shown. The erroneous directive is ignored.	
`#pragma fini foo`	

`ignoring malformed #pragma init`	
Type: Warning	*Options:* All
The compiler encountered a `#pragma init` directive that did not have the form shown. The erroneous directive is ignored.	
`#pragma init foo`	

`ignoring malformed #pragma int_to_unsigned symbol`	
Type: Warning	*Options:* All
The compiler encountered a `#pragma int_to_unsigned` directive that did not have the form shown. The erroneous directive is ignored.	
`#pragma int_to_unsigned strlen();`	

`ignoring malformed #pragma weak symbol [=value]`	
Type: Warning	*Options:* All
The compiler encountered a `#pragma` weak directive that did not have the form shown. The erroneous directive is ignored.	
`#pragma weak write,_write`	

`implicitly declaring function to return int:` *name*`()`	
Type: Warning	*Options:* -v
The program calls function which has not been previously declared. The compiler warns you that it is assuming that function *name* returns int.	

`void v(void){`
`  g();`
`}`	

`improper cast of void expression`	
Type: Error	*Options:* All
You cannot cast a `void` expression to something other than `void`.	

`f(void){`
`  void v(void);`
`  int i = (int) v();`
`}`	

`improper member use:` *name*	
Type: Warning, Error	*Options:* All
Your program contains an expression with a `->` or `.` operator, and *name* is not a member of the structure or union that the left side of the operator refers to, but it is a member of some other structure or union. This diagnostic is an error if the member is not "unique." A unique member is part of one or more structures or unions but has the same type and offset in all of them.	

```
struct s1 { int x,y; };
struct s2 { int q,r; };
f(void){
    struct s1 *ps1;
    ps1->r = 3;
}
```

`improper pointer subtraction`	
Type: Warning, Error	*Options:* All
The operands of a subtraction are both pointers, but they point at different types. You may only subtract pointers of the same type that point to the same array. The diagnostic is a warning if the pointers point to objects of the same size, and an error otherwise.	

```
f(void){
    int *ip;
    char *cp;
    int i = ip - cp;
}
```

`improper pointer/integer combination: arg #n`	
Type: Warning	*Options:* All
At a function call for which there is a function prototype declaration in scope, the code is passing an integer where a pointer is expected, or vice versa.	

```
int f(char *);
g(void){
    f(5);
}
```

`improper pointer/integer combination: op "operator"`	
Type: Warning	*Options:* All
One of the operands of *operator* is a pointer and the other is an integer, but this combination is invalid.	

```
f(void){
    int i = "abc";
    int j = i ? 4 :  "def";
}
```

`inappropriate qualifiers with "void"`	
Type: Warning	*Options:* All
You may not qualify `void` (with `const` or `volatile`) when it stands by itself.	

```
int f(const void);
```

`#include <... missing '>'`	
Type: Warning	*Options:* All
In a `#include` directive for which the header name began with <, the closing > character was omitted.	
`#include <stdio.h`	

`#include directive missing file name`	
Type: Error	*Options:* All
A `#include` directive did not specify a file to include.	
`#include`	

`#include of /usr/include/... may be non-portable`	
Type: Warning	*Options:* All
The source file included a file with the explicit prefix `/usr/include`. Such an inclusion is implementation-dependent and non-portable. On some systems the list of default places to look for a header might not include the `/usr/include` directory. In such a case the wrong file might be included.	
`#include </usr/include/stdio.h>`	

`incomplete #define macro parameter list`	
Type: Error	*Options:* All
In the definition of a function-like parameter, the compiler did not find a) character on the same (logical) line as the `#define` directive.	
`#define mac(a`	

`incomplete struct/union/enum` *tag*`:` *name*	
Type: Error	*Options:* All
You declared an object *name*, with struct, union, or enum type and tag *tag*, but the type is incomplete.	
`struct s st;`	

`inconsistent redeclaration of extern:` *name*	
Type: Warning	*Options:* All
You have redeclared function or object *name* with storage class extern for which there was a previous declaration that has since gone out of scope. The second declaration has a type that conflicts with the first.	

```
f(void){
    int *p = (int *) malloc(5*sizeof(int));
}
g(void){
    void *malloc();
}
```

`inconsistent redeclaration of static:` *name*	
Type: Warning	*Options:* All

You have redeclared an object or function that was originally declared with storage class static. The second declaration has a type that conflicts with the first.

The two most frequent conditions under which this diagnostic may be issued are:

1. A function was originally declared at other than file scope and with storage class static. The subsequent declaration of the function has a type that conflicts with the first.

2. A function or object was originally declared at file scope and with storage class static. A subsequent declaration of the same object or function at other than file scope used storage class extern (or possibly no storage class, if a function), and there was an intervening, unrelated, declaration of the same name.

```
f(void){
    static int myfunc(void);
}
g(void){
    static char *myfunc(void);
}
static int x;
f(void){
    int x;              /* unrelated */
{
        extern float x;  /* related to first declaration */
    }
}
```

`inconsistent storage class for function:` *name*	
Type: Warning	*Options:* All
ANSI C requires that the first declaration of a function or object at file scope establish its storage class. You have redeclared function *name* in an inconsistent way according to these rules.	

`g(void){`
`  int f(void);`
`  static int f(void);`
`}`	

`initializer does not fit:` *value*	
Type: Warning	*Options:* All
The value *value* does not fit in the space provided for it. That is, if it were fetched from that space, it would not reproduce the same value as was put in. In the message, *value* is represented as a hexadecimal value if the initializer is unsigned, decimal if it is signed. The hexadecimal values 0x80 through 0xff will not fit into a char, 0x8000 through 0xffff will not fit into a short, and 0x80000000 through 0xffffffff will not fit into an int, but these values will work with their corresponding unsigned types.	
`struct s {signed int m1:3; unsigned int m2:3;} st = {4, 5};` `unsigned char uc = 300u;`	

`integer overflow detected: op "`*operator*`"`	
Type: Warning	*Options:* All
The compiler attempted to compute the result of an *operator* expression at compile-time, and determined that the result would overflow. The low-order 32 bits of the result are retained, and the compiler issues this diagnostic.	
`int i = 1000000 * 1000000;`	

`integral constant expression expected`	
Type: Warning	*Options:* All
The compiler expected (required) an integral constant or an expression that can be evaluated at compile time to yield an integral value. The expression you wrote contained either a non-integral value, a reference to an object, or an operator that cannot be evaluated at compile time.	
`int ia[5.0];`	

`integral constant too large`	
Type: Warning	*Options:* All
An integral constant is too large to fit in an `unsigned long`.	
`int i = 12345678901234567890;`	

`internal compiler error:` *message*	
Type: Warning	*Options:* All
This message does not diagnose a user programming error (usually), but rather a problem with the compiler itself. One of the compiler's internal consistency checks has failed. The problem diagnosed by *message* is important to our support staff but is probably meaningless to you. You can help us to identify the problem by performing the following and then calling a support center. Run the `cc` command again with the same options as when it failed, plus the `-P` option. You will not get the internal compiler error message again. However, assuming you compiled *file*`.c`, the cc command will create a *file*`.i` file in your current directory. This file will help us to identify the compiler problem.	

`interpreted as a #line directive`	
Type: Warning	*Options:* -Xc
A source line was encountered that had a number where the directive name usually goes. Such a line is reserved for the compiler's internal use, but it must be diagnosed in the `-Xc` (strictly conforming) mode.	
`# 9`	

`invalid cast expression`	
Type: Error	*Options:* All
You cannot apply the cast to the expression because the types are unsuitable for casting. Both the type of the expression being cast and the type of the cast must be scalar types. A pointer may only be cast to or from an integral type.	

`f(void){`
`  struct s {int x;} st;`
`  int i = (int) st;`
`}`	

`invalid compiler control line in ".i" file`	
Type: Error	*Options:* All
A `.i` file, the result of a `cc -P` command, is assumed to be a reserved communication channel between the preprocessing phase and the compilation phase of the compiler. The `.i` file lets you examine that intermediate form to detect errors that may otherwise be hard to detect. However, the compiler expects to find only a few directives that are used for internal communication. The source file that was compiled (a `.i` file) contained a preprocessing directive other than one of the special directives.	

`invalid directive`	
Type: Error	*Options:* All
The identifier that follows a # in a preprocessing directive line was one that the compiler did not recognize.	
`# unknown foo`	

`invalid multibyte character`	
Type: Error	*Options:* All
A multibyte character in a string literal or character constant could not be converted to a single wide character in the host environment.	

`invalid source character: 'c'`	
Type: Error	*Options:* -Xa, -Xc
The compiler encountered a character in the source program that is not a valid ANSI C token.	
`int i = 1$;`	

`invalid source character: <hex value>`	
Type: Error	*Options:* All
This message diagnoses the same condition as the previous one, but the invalid character is not printable. The *hex~value* between the brackets in the diagnostic is the hexadecimal value of the character code.	

`invalid switch expression type`	
Type: Error	*Options:* All
The controlling expression of a switch statement could not be converted to `int`. This message always follows switch expression must have integral type .	

```
f(void){
    struct s {int x;} sx;
    switch(sx){
    case 4:  ;
    }
}
```

`invalid token:` *non-token*	
Type: Error	*Options:* All
The compiler encountered a sequence of characters that does not comprise a valid token. An invalid token may result from the preprocessing `##` operator. The offending *non-token* is shown in the diagnostic. If the *non-token* is longer than 20 characters, the first 20 are printed, followed by ". . .". The offending invalid token is ignored.	

```
#define PASTE(1,r) 1 ## r
double d1 = 1e;
double d2 = PASTE(1,e);
int i = 1veryverylongnontoken;
```

`invalid token in #define macro parameters:` *token*	
Type: Error	*Options:* All
The compiler encountered an inappropriate token while processing the argument list of a function-like macro definition. *token* is the erroneous token.	

```
#define mac(a,4) a b c
```

`invalid token in directive`	
Type: Error	*Options:* All
The compiler found an invalid token at the end of what would otherwise be a correctly formed directive.	
`#line 7 "file.c`	

`invalid type combination`	
Type: Error	*Options:* All
You used an inappropriate combination of type specifiers in a declaration.	
`short float f;`	

`invalid type for bit-field:` *name*	
Type: Error	*Options:* All
The type you chose for bit-field *name* is not permitted for bit-fields. Bit-fields may only be declared with integral types.	
`struct s { float f:3; };`	

`invalid use of "defined" operator`	
Type: Error	*Options:* All
A defined operator in a `#if` or `#elif` directive must be followed by an identifier or ( )'s that enclose an identifier. The source code did not use it that way.	

`#if defined`
`int i = 1;`
`#endif`	

`invalid white space character in directive`	
Type: Warning	*Options:* All
The only white space characters that are permitted in preprocessing directives are space and horizontal tab. The source code included some other white space character, such as form feed or vertical tab. The compiler treats this character like a space.	

`label redefined:` *name*	
Type: Error	*Options:* All
The same label *name* has appeared more than once in the current function. (A label's scope is an entire function.)	

```
f(void){
    int i;
    i = 1;
    if (i) {
L:
        while (i)
            g();
        goto L;
    }
L:  ;
}
```

`left operand must be modifiable lvalue: op "`*operator*`"`	
Type: Error	*Options:* All
The operand on the left side of *operator* must be a modifiable lvalue, but it wasn't.	

```
f(void){
    int i = 1;
    +i -= 1;
}
```

`left operand of "->" must be pointer to struct/union`	
Type: Warning, Error	*Options:* All
The operand on the left side of a `->` operator must be a pointer to a structure or union, but it wasn't. The diagnostic is a warning if the operand is a pointer, an error otherwise.	

```
struct s { int x; };
f(void){
    long *lp;
    lp->x = 1;
}
```

`left operand of "." must be lvalue in this context`	
Type: Warning	*Options:* All
The operand on the left side of a `.` operator is an expression that does not yield an lvalue. Usually this results from trying to change the return value of a function that returns a structure.	

```
struct s { int ia[10]; };
struct s sf(void);
f(void){
    sf().ia[0] = 3;
}
```

`left operand of "." must be struct/union object`	
Type: Warning, Error	*Options:* All
The `.` operator is only supposed to be applied to structure or union objects. The diagnostic is an error if the operand to the left of `.` is an array, pointer, function call, enumeration constant or variable, or a register value that got allocated to a register; it is a warning otherwise.	

`f(void){`
`  struct s { short s; };`
`  int i;`
`  i.s = 4;`
`}`	

`()-less function definition`	
Type: Error	*Options:* All
The declarator portion of a function definition must include parentheses. You cannot define a function by writing a typedef name for a function type, followed by an identifier and the braces that define a function.	
`typedef int F();` `F f{ }`	

`loop not entered at top`	
Type: Warning	*Options:* All
The controlling expression at the beginning of a `for` or `while` loop cannot be reached by sequential flow of control from the statement before it.	

```
f(void){
    int i;
    goto lab;
    for (i = 1; i > 0; --i) {
lab:;
        i=5;
    }
}
```

`macro recursion`	
Type: Fatal	*Options:* `-Xt`
The source code calls a macro that calls itself, either directly or indirectly. ANSI C's semantics prevent further attempts to rescan the macro. Older C compilers would try to rescan the macro, which eventually leads to a fatal error. Because the rescanning rules are different for ANSI C and its predecessor, the ANSI C compiler provides the old behavior in `-Xt` mode, which includes producing this diagnostic when macro recursion is detected.	

```
#define a(x) b(x)
#define b(x) a(x)
a(3)
```

`macro redefined:` *name*	
Type: Warning	*Options:* All
The source code redefined a macro. Previous releases of K&R C allowed such redefinitions silently if both definitions were identical except for the order and spelling of formal parameters. ANSI C requires that, when a macro is redefined correctly, the definitions must be identical including the order and spelling of formal parameters. This diagnostic is produced under all options if the new macro definition disagrees with the old one. For strict conformance, it is also produced under the `-Xc` option when the macro definitions disagree only in the spelling of the formal parameters.	
`#define TIMES(a,b) a * b` `#define TIMES(a,b) a - b`	

`macro replacement within a character constant`	
Type: Warning	*Options:* -Xt

Previous releases of K&R C allowed the value of a formal parameter to be substituted in a character constant that is part of a macro definition. ANSI C does not permit such a use.
 	
The proper way to express this construct in ANSI C is the following:
 	
`#define /* form control character */`
`int ctrl_c = CTRL(*'c*');`	

`#define CNTRL(x) ('x'&037) /* form control character */` `int ctrl_c = CTRL(c);`	

`macro replacement within a string literal`	
Type: Warning	*Options:* `-Xt`

This message diagnoses a similar condition to the preceding one, except the substitution is being made into a string literal.
 	
ANSI C provides a way to accomplish the same thing. The # "string-ize" operator turns the tokens of a macro argument into a string literal, and adjacent string literals are concatenated. The correct form is:
 	
`#define HELLO(name) name`
`char *hello_mindy = HELLO("Mindy");`	

`#define HELLO(name) "name"` `char *hello_mindy = HELLO(Mindy);`	

`member cannot be function:` *name*	
Type: Error	*Options:* All
A function may not be a member of a structure or union, although a pointer to a function may. You declared member *name* as a function.	
`struct s { int f(void); };`	

`mismatched "?" and ":"`	
Type: Error	*Options:* All
An expression in a `#if` or `#elif` directive contained a malformed ?~: expression.	

`#if defined(foo) ? 5`
`  int i;`
`#endif`	

`mismatched parentheses`	
Type: Error	*Options:* All
Parentheses were mismatched in a preprocessing conditional compilation directive.	

```
#if ((1)
    int i = 1;
#endif
```

`missing ")"`	
Type: Error	*Options:* All
In a test of a predicate that follows a `#` operator in a `#if` or `#elif` directive, the `)` that follows the assertion was missing.	

```
#if # system(unix
char *system = "unix";
#endif
```

`missing operand`	
Type: Error	*Options:* All
The constant expression of a preprocessing conditional compilation directive is malformed. An expected operand for some operator was missing.	

```
#define EMPTY
#if EMPTY / 4
    int i = 1;
#endif
```

`missing operator`	
Type: Error	*Options:* All
The constant expression of a preprocessing conditional compilation directive is malformed. An operator was expected but was not encountered.	

`#if 1 4`
`    int i = 1;`
`#endif`	

`missing tokens between parentheses`	
Type: Error	*Options:* All
In a `#assert` directive, there are no assertions within the parentheses of the predicate.	
`#assert system()`	

`modification of typedef with "`*modifier*`" ignored`	
Type: Warning	*Options:* `All`
You are applying a type *modifier* to a typedef name, which ANSI C prohibits. ANSI C only permits you to modify a typedef with a type qualifier.	
`typedef int INT;` `unsigned INT i`	

`modulus by zero`	
Type: Warning, Error	*Options:* All
The second operand of a `%` operator is zero. If the modulus operation is part of a #if or `#elif` directive, the result is taken to be zero. The diagnostic is a warning if the modulus is in executable code, an error otherwise.	

`#if 42 % 0`
`  int i = 1;`
`#endif`	

`more than one character honored in character constant:` *constant*	
Type: Warning	*Options:* All
A character constant has an integral value that derives from the character codes of the characters. If a character constant comprises more than one character, the encoding of the additional characters depends on the implementation. This warning alerts you that the encoding that the preprocessing phase uses for the character constant *constant* is different in this release of the C compiler from the one in previous releases, which only honored the first character. (The encoding for character constants you use in executable code is unchanged.)	

`#if 'ab' != ('b' * 256 + 'a')`
`#error unknown encoding`
`#endif`	

`"#" must be followed by formal identifier in #define`	
Type: Error	*Options:* All
The "string-ize" operator `#` must be followed by the name of a formal parameter in a function-like macro.	
`#define mac(a) # + a`	

must have type "function-returning-unsigned": *name*	
Type: Warning	*Options:* All
The *name* that is a part of a `#pragma int_to_unsigned` directive must be an identifier whose type is function-returning-`unsigned`.	
`extern int f(int);` `#pragma int_to_unsigned f`	

newline in character constant	
Type: Error	*Options:* All
You wrote a character constant that had no closing *'* on the same line as the beginning *'*.	
`int i = 'a` `;`	

newline in string literal	
Type: Warning, Error	*Options:* All
You wrote a string literal that had no closing " (quote marks) on the same line as the beginning ". The diagnostic is a warning if the string literal is part of a preprocessing directive (and the compiler provides the missing ") and an error otherwise.	
`char *p = "abc` `;`	

`newline not last character in file`	
Type: Warning	*Options:* All
Every non-empty source file and header must consist of complete lines. This diagnostic warns that the last line of a file did not end with a newline.	

`no file name after expansion`	
Type: Error	*Options:* All
You used the form of `#include` directive that permits macro expansion of its argument, but the resulting expansion left no tokens to be taken as a file name.	
`#define EMPTY` `#include EMPTY`	

`no hex digits follow \x`	
Type: Warning	*Options:* `-Xa, -Xc`
The `\x` escape in character constants and string literals introduces a hexadecimal character escape. The `\x` must be followed by at least one hexadecimal digit.	
`char *cp = "\xz";`	

`no macro replacement within a character constant`	
Type: Warning	*Options:* `-Xa, -Xc`
This message is the inverse of macro replacement within a character constant . It informs you that the macro replacement that was done for –Xt mode is not being done in `-Xa` or `-Xt` mode.	

no macro replacement within a string literal	
Type: Warning	*Options:* `-Xa, -Xc`
This message is the inverse of macro replacement within a string literal . It informs you that the macro replacement that was done for `-Xt` mode is not being done in `-Xa` or `-Xt` mode.	

no tokens after expansion	
Type: Error	*Options:* All
After macro expansion was applied to the expression in a `#line` directive, there were no tokens left to be interpreted as a line number.	
`#define EMPTY` `#line EMPTY`	

no tokens follow "#pragma"	
Type: Warning	*Options:* `-v`
The compiler encountered a `#pragma` directive that contained no other tokens.	
`#pragma`	

no tokens following "#assert name ("	
Type: Error	*Options:* All
A use of the `#assert` directive is malformed. The assertions and the `)` that should follow are missing.	
`#assert system(`	

`no tokens in #line directive`	
Type: Error	*Options:* All
The rest of a `#line` directive was empty; the line number and optional file name were missing.	
`#line`	

`non-constant initializer: op "operator"`	
Type: Error	*Options:* All
The initializer for an extern, static or array object must be a compile-time constant. The initializers for an automatic structure or union object, if enclosed in { } , must also be compile-time constants. *operator* is the operator whose operands could not be combined at compile time.	
`int j;` `int k = j+1;`	

`non-formal identifier follows "#" in #define`	
Type: Warning	*Options:* All
The identifier that follows a `# operator` in a macro definition must be a formal parameter of a function-like macro.	
`#define mac(a) "abc" # b`	

non-integral case expression	
Type: Error	*Options:* All
The operand of a `case` statement must be an integral constant.	

```
f(void){
    int i = 1;
    switch (i) {
    case 5.0:  ;
    }
}
```

non-unique member requires struct/union: *name*	
Type: Error	*Options:* All
The operand on the left side of a . operator was not a structure, union, or a pointer to one, and member *name* was not unique among all structure and union members that you have declared. You should only use . with structures or unions, and the member should belong to the structure or union corresponding to the left operand.	

```
struct s1 { int x,y; };
struct s2 { int y,z; };
f(void){
    long *lp;
    lp.y = 1;
}
```

non-unique member requires struct/union pointer: *name*	
Type: Error	*Options:* All
This message diagnoses the same condition as the preceding one, but for the -> operator.	

`null character in input`	
Type: Error	*Options:* All
The compiler encountered a null character (a character with a character code of zero).	

`null dimension:` *name*	
Type: Warning, Error	*Options:* All
A dimension of an array is null in a context where that is prohibited. The diagnostic is a warning if the offending dimension is outermost and an error otherwise.	

`int ia[4][];`
`struct s { int x, y[]; };`
`int i = sizeof(int []);`	

`number expected`	
Type: Error	*Options:* All
The compiler did not find a number where it expected to find one in a #if or #elif directive.	

`#if 1 +`
`  int i = 1;`
`#endif`	

`old-style declaration hides prototype declaration:` *name*	
Type: Warning	*Options:* `-v`
You redeclared function *name* in an inner scope. The outer declaration was a function prototype declaration, but the inner one lacks parameter information. By ANSI C's scoping rules, the parameter information is hidden and the automatic conversions of types that the prototype would have provided are suppressed.	

```
extern double sin(double);
f(void){
    extern double sin();
    double d;
    d = sin(1);/* Note: no conversion to double! */
}
```

`old-style declaration; add "int"`	
Type: Warning	*Options:* All
Objects and functions that are declared at file scope must have a storage class or type specifier. You will get this warning if you omit both.	

```
i;
f(void);
```

`only one storage class allowed`	
Type: Error	*Options:* All
You specified more than one storage class in a declaration.	

```
f(void){
register auto i;
}
```

only qualifiers allowed after *	
Type: Error	*Options:* All
You may only specify the const or volatile type qualifiers after a * in a declaration.	

```
int * const p;
int * unsigned q;
```

only "register" valid as formal parameter storage class	
Type: Error	*Options:* All
You may specify a storage class specifier in a function prototype declaration, but only register is permitted.	

```
int f(
    register int x,
    auto int y
);
```

operand cannot have void type: op "*operator*"	
Type: Error	*Options:* All
One of the operands of *operator* has `void` type.	

```
f(void){
void v(void);
int i = v();
}
```

`operand must be modifiable lvalue: op "operator"`	
Type: Error	*Options:* All
The operand of *operator* must be a modifiable lvalue, but it wasn't.	

```
f(void){
int i = --3;
}
```

`operand treated as unsigned: constant`	
Type: Warning	*Options:* `-Xt`
An operand you used in a `#if` or `#elif` directive has a value greater than `LONG_MAX` (2147483647) but has no unsigned modifier suffix or U). Previous releases of K&R C treated such *constant*s as signed quantities which, because of their values, actually became negative. ANSI C treats such constants as unsigned long integers, which may affect their behavior in expressions. This diagnostic is a transition aid that informs you that the value is being treated differently from before.	

```
#if 2147483648 > 0
    char *mesg = "ANSI C-style";
#endif
```

`operands have incompatible pointer types: op "operator"`	
Type: Warning	*Options:* All
You have applied *operator* to pointers to different types.	

```
f(void){
    char *cp;
    int *ip;
    if (ip < cp)
        ;
}
```

`operands have incompatible types: op "operator"`	
Type: Error	*Options:* All
The types of the operands for *operand* are unsuitable for that kind of operator.	

```
f(void){
    char *cp;
    int *ip;
    void *vp = ip + cp;
}
```

`operands must have` *category* type: op *"operator"*	
Type: Error	*Options:* All
The operands for *operator* do not fall into the appropriate category for that operator. *category* may be arithmetic, integral, or scalar.	

```
f(void){
    int ia[5];
    int *ip = ia/4;
}
```

`out of scope extern and prior uses redeclared as static:` *name*	
Type: Warning	*Options:* `-Xc, -v`
You declared *name* as extern in a block that has gone out of scope. Then you declared *name* again, this time as static. The ANSI C compiler treats the object or function as if it were static, and all references, including ones earlier in the source file, apply to the static version.	

```
f(void){
    extern int i;
}
static int i;
```

`overflow in hex escape`	
Type: Warning	*Options:* All
In a hexadecimal escape (`\x`) in a character constant or string literal, the accumulated value for the escape grew too large. Only the low-order 32 bits of value are retained.	
`int i = '\xabcdefedc';`	

`parameter mismatch:` *n-decl* declared, *n-def* defined	
Type: Warning	*Options:* All
A function prototype declaration and an old-style definition of the function disagree in the number of parameters. The declaration had *n-decl* parameters, while the definition had *n-def*.	

`int f(int);`
`int f(i,j)`
`int i,j;`
`{}`	

`parameter not in identifier list:` *name*	
Type: Error	*Options:* All
Variable *name* appears in an old-style function definition's parameter declarations, but it does not appear in the parameter identifier list.	

`f(a,b)`
`int i;`
`{}`	

`parameter redeclared:` *name*	
Type: Error	*Options:* All
You have used *name* more than once as the name for a parameter in a function definition.	

```
int f(int i, int i) { }
int g(i,j)
int i;
int i;
{ }
```

`prototype mismatch:` *n1* arg*[s]* passed, *n2* expected	
Type: Error	*Options:* All
You called a function for which there is a function prototype declaration in scope, and the number of arguments in the call, did not match the number of parameters in the declaration, *n1*.	

```
int f(int);
g(void){
    f(1,2);
}
```

`return value type mismatch`	
Type: Error	*Options:* All
You are attempting to return a value from a function that cannot be converted to the return-type of the function.	

```
f(void){
    struct s { int x; } st;
    return( st );
}
```

`semantics of "operator" change in ANSI C; use explicit cast`	
Type: Warning	*Options:* All

The type promotion rules for ANSI C are slightly different from those of previous versions of K&R C. In the current release the default behavior is to duplicate the previous rules. In future releases the default will be to use ANSI C rules. You may obtain the ANSI C interpretation by using the –Xa option for the cc command.

Previous K&R C type promotion rules were "unsigned-preserving." If one of the operands of an expression was of unsigned type, the operands were promoted to a common unsigned type before the operation was performed.

ANSI C uses "value-preserving" type promotion rules. An unsigned type is promoted to a signed type if all its values may be represented in the signed type.

The different type promotion rules may lead to different program behavior for the operators that are affected by the unsigned-ness of their operands:

The division operators: `/, /=, %, %=`.

The right shift operators: `>>, >>=`.

The relational operators: `<, <=, >, >=`.

The warning message tells you that your program contains an expression in which the behavior of *operator* will change in the future. You can guarantee the behavior you want by inserting an explicit cast in the expression.

You can get the same behavior as in previous versions of K&R C by adding an explicit cast:

```
f(void){
    unsigned char uc;
    int i;
    /* was unsigned divide in K&R C, signed in ANSI C */
    i /= (unsigned int) uc;
}
```

```
f(void){
    unsigned char uc;
    int i;
    /* was unsigned divide in K&R C, signed in ANSI C */
    i /= uc;
}
```

`shift count negative or too big:` *op n*	
Type: Warning	Options: All
The compiler determined that the shift count (the right operand) for shift operator *op* is either negative or bigger than the size of the operand being shifted.	

```
f(){
    short s;
    s <<= 25;
}
```

`statement not reached`	
Type: Warning	*Options:* All
This statement in your program cannot be reached because of `goto`, `break`, `continue`, or `return` statements preceding it.	

```
f(void){
    int i;
    return i;
    i = 4;
}
```

`static function called but not defined:` *name*`()`	
Type: Warning	*Options:* All
The program calls function *name*, which has been declared static, but no definition of *name* appears in the translation unit. (The line number that is displayed in the message is one more than the number of lines in the file, because this condition can be diagnosed only after the entire translation unit has been seen.)	

```
static int statfunc(int);
void
f(){
    int i = statfunc(4);
}
```

`static redeclares external:` *name*	
Type: Warning	*Options:* All
You reused *name* as the name of a static object or function after having used it in the same block as the name of an extern object or function. The version of *name* that remains visible is the static version.	

`f(void){`
`  extern int i;`
`  static int i;`
`}`	

`storage class after type is obsolescent`	
Type: Warning	*Options:* `-v`
According to the ANSI C standard, writing declarations in which the storage class specifier is not first is "obsolescent."	
`int static i;`	

`storage class for function must be static or extern`	
Type: Warning	*Options:* All
You used an inappropriate storage class specifier for a function declaration or definition. Only extern and static may be used, or the storage class may be omitted. The specifier is ignored.	

`f(void){`
`  auto g(void);`
`}`	

`string literal expected after #file`	
Type: Error	*Options:* All
The `#file` directive (which is reserved for the compilation system) is used for internal communication between preprocessing and compilation phases. A string literal operand is expected as the operand.	

`string literal expected after #ident`	
Type: Error	*Options:* All
A `#ident` directive must be followed by a normal (not wide character) string literal.	
`#ident no-string`	

`string literal expected after #line <number>`	
Type: Warning	*Options:* All
This diagnostic is similar to `string literal expected after # <number>`, except that it applies to the standard `#line` directive.	

`string literal must be sole array initializer`	
Type: Warning	*Options:* All
You may not initialize a character array with both a string literal and other values in the same initialization.	
`char ca[] = { "abc", 'd' };`	

`struct/union has no named members`	
Type: Warning	*Options:* All
You have declared a structure or union in which none of the members is named.	
`struct s { int :4; char :0; };`	

`struct/union-valued initializer required`	
Type: Error	*Options:* All
ANSI C allows you to initialize an automatic structure or union, but the initializer must have the same type as the object being initialized.	

`f(void){`
`  int i;`
`  struct s { int x; } st = i;`
`}`	

`switch expression must have integral type`	
Type: Warning, Error	*Options:* All
You wrote a `switch` statement in which the controlling expression did not have integral type. The message is a warning if the invalid type is a floating-point type and an error otherwise. A floating-point switch expression is converted to `int`.	

`f(void){`
`  float x;`
`  switch (x) {`
`  case 4:  ;`
`  }`
`}`	

`syntax error before or at:` *token*	
Type: Error	*Options:* All
This is an all-purpose diagnostic that means you have juxtaposed two (or more) language tokens inappropriately. The compiler shows you the *token* at which the error was detected.	

`f(void){`
`    int i = 3+;`
`}`	

`syntax error in macro parameters`	
Type: Error	*Options:* All
The macro parameter list part of a function-like macro definition is malformed. The list must be a comma-separated list of identifiers and was not.	
`#define mac(a,b,) a b`	

`syntax error, probably missing ",", ";" or "="`	
Type: Error	*Options:* All
You wrote a declaration that looked like a function definition, except that the type of the symbol declared was not "function returning." You probably left out a `L.` `;` or `=`.	
`int i` `int j;`	

`syntax error: empty declaration`	
Type: Warning	*Options:* All
You wrote a null statement at file scope. This looks like an empty declaration statement. K&R C permitted this previously, but ANSI C does not.	
`int i;;`	

`syntax error: "&..." invalid`	
Type: Warning	*Options:* `-Xc`
You wrote `&...` in a program that was compiled with the `-Xc` option. `&...` is invalid ANSI C syntax. You should not use this notation explicitly.	

`syntax requires ";" after last struct/union member`	
Type: Warning	*Options:* All
You omitted the `;` that C syntax requires after the last structure or union member in a structure or union declaration.	
`struct s { int x };`	

(*type*) `tag redeclared:` *name*	
Type: Error	*Options:* All
You have redeclared tag *name* that was originally a *type* tag.	
`struct q { int m1, m2; };` `enum q { e1, e2 };`	

`token not allowed in directive:` *token*	
Type: Error	*Options:* All
You used a *token* in a `#if` or `#elif` directive that is neither a valid operator for constant expressions, nor a valid integer constant.	

`#if 1 > "1"`
`    int i = 1;`
`#endif`	

`token-less macro argument`	
Type: Warning	*Options:* `-Xc`
The actual argument to a preprocessor macro consisted of no tokens. The ANSI C standard regards this condition as undefined. The C compiler treats the empty list of tokens as an empty argument, and, under the `-Xc` mode, it also issues this warning.	
`#define m(x) x+3` `int i = m();`	

`tokens after -A- are ignored`	
Type: Warning	*Options:* All
In the `-A-` option to the `cc` command, there were additional tokens adjacent to the option. They are ignored.	
`cc -A-extra -c x.c`	

`tokens expected after "# identifier ("`	
Type: Error	*Options:* All
When the # operator is used in a `#if` or `#elif` directive to select a predicate instead of a like-named macro, the predicate must be followed by a parenthesized list of tokens.	

```
#if #system(
    char *system = "unix";
#endif
```

`tokens expected after "("`	
Type: Error	*Options:* All
In a `#unassert` directive, the assertion(s) and closing) after the predicate were missing.	

```
#unassert system(
```

`tokens expected between parentheses`	
Type: Error	*Options:* All
The name of an assertion of a predicate to test was omitted in an `#if` or `#elif` directive.	

```
#if #system()
    char *sysname = "??";
#endif
```

`tokens ignored after "-U{identifier}"`	
Type: Warning	*Options:* All
In the command line `-U` option, there were tokens following the name of the macro to be undefined.	
`cc -Uunix,u3b2 -c x.c`	

`tokens ignored at end of directive line`	
Type: Warning	*Options*: All
A directive line contains extra tokens that are not expected as part of the directive.	
`#undef a b/* can only undefine one */`	

`too many array initializers`	
Type: Error	*Options:* All
You provided more initializers for an array than the array can hold.	
`int ia[3] = { 1, 2, 3, 4 };`	

too many #else's	
Type: Warning	*Options:* All
The code contained more than one `#else` directive in a preprocessing `if`-section. All `#else` directives after the first are taken to be false.	

```
#ifdef ONE
    int i = 1;
#else
    int i = 2;
#else
    int i = 3
#endif
```

too many errors	
Type: Fatal	*Options:* All
The compiler encountered too many errors to make further processing sensible. Rather than produce further diagnostics, the compiler exits.	

too many initializers for scalar	
Type: Error	*Options:* All
A `{ }` -bracketed initialization for a scalar contains more than one value.	

```
int i = { 1, 2 };
```

`too many struct/union initializers`	
Type: Error	*Options:* All
You have provided too many initializers for a structure or union.	
`struct s { int x,y; } st = { 1,2,3 };`	

`trailing "," prohibited in enum declaration`	
Type: Warning	*Options:* -Xc, -v
You supplied an extra comma at the end of an enumeration type declaration. The extra comma is prohibited by the syntax.	
`enum e { e1, e2, };`	

`trigraph sequence replaced`	
Type: Warning	*Options:* `-Xt`
ANSI C introduces the notion of trigraphs, three-character sequences that stand for a single character. All such sequences begin with `??`. Because sequences that are interpreted as trigraphs may appear in existing code, the K&RC compiler produces a transitional diagnostic when such sequences are encountered.	
`char *surprise = "this is a trigraph??!";`	

`type does not match prototype:` *name*	
Type: Warning	*Options:* All

You provided a function prototype declaration for a function, but used an old-style definition. The type for parameter *name* in that definition is incompatible with the type you used in the prototype declaration.

The following example shows an especially confusing instance of this diagnostic.

```
int f(char);
int f(c)
char c;
{ }
```

`f` has an old-style definition. For compatibility reasons, `f`'s arguments must therefore be promoted according to the default argument promotions, which is how they were promoted before the existence of function prototypes. Therefore, the value that must actually be passed to `f` is an `int`, although the function will only use the `char` part of the value. The diagnostic, then, identifies the conflict between the int that the function expects and the `char` that the function prototype would (conceptually) cause to be passed.

There are two ways to fix the conflict:

1. Change the function prototype to read `int f(int);`

2. Define f with a function prototype definition:

```
int f(char);
int f(char c)
{}
```

```
int f(char *);
int f(p)
int *p;
{}
```

`typedef already qualified with "qualifier"`	
Type: Warning	*Options:* All
A type specifier includes a typedef and an explicit type qualifier, *qualifier*. The typedef already included *qualifier* when it was declared.	
`typedef volatile int VOL;` `volatile VOL v;`	

`typedef declares no type name`	
Type: Warning	*Options:* All
In a declaration with storage class typedef, no type name was actually declared. This is probably a programming error.	
`typedef struct s { int x; };`	

`typedef redeclared:` *name*	
Type: Warning	*Options:* All
You have declared typedef *name* more than once. The later declaration has an identical type to the first.	
`typedef int i;` `typedef int i;`	

`typedef redeclares external:` *name*	
Type: Warning	*Options:* All
You declared typedef *name*, but there is an extern of the same name in the same block. The typedef hides the external.	

```
f(void){
    extern int INT;
    typedef int INT;
}
```

`"typedef" valid only for function declaration`	
Type: Warning	*Options:* All
A function definition may not have the typedef storage class. It is ignored here.	

```
typedef int f(void){}
```

`-U option argument not an identifier`	
Type: Error	*Options:* All
An identifier must follow the `-U cc` command line option.	

```
cc -U3b2 -c x.c
```

`unacceptable operand for unary &`	
Type: Error	*Options:* All
You attempted to take the address of something whose address cannot be taken.	

```
f(void){
    int *ip = &g();
}
```

`#unassert requires an identifier token`	
Type: Error	*Options:* All
The #unassert directive must name a predicate to "un-assert."	
`#unassert 5`	

`undefined label:` *label*	
Type: Error	*Options:* All
You wrote a `goto` in the current function, but you never defined the target *label* anywhere within the function.	

`f(void){`
`  goto L;`
`}`	

`undefined struct/union member:` *name*	
Type: Error	*Options:* All
Your program made reference to a structure or union member, that has not been declared as part of any structure.	

`struct s { int x; };`
`f(void){`
`  struct s q;`
`  q.y = 1;`
`}`	

`undefined symbol:` *name*	
Type: Error	*Options:* All
You referred to symbol *name* for which there is no declaration in scope.	

```
f(void){
g(i);
}
```

`undefining __STDC__`	
Type: Warning	Options: `-Xt`
ANSI C prohibits undefining the predefined symbol `__STDC__`. However, this release of the C compiler permits you to do so in transition mode (only). You may want to use this feature to test C code that you have written to work in both an ANSI C and non-ANSI C environment. For example, suppose you have C code that checks `__STDC__`, declaring function prototype declarations if it is defined, and old-style function declarations (or definitions) if not. Because the C compiler predefines `__STDC__`, you would ordinarily be unable to check the old-style code, and you would have to run the code through another (non-ANSI C) compiler. By undefining `__STDC__` (usually on the command line), you can use the C compiler to do the checking. This diagnostic tells you, as required, that you are violating ANSI C constraints.	

```
#undef __STDC__/*usually -U__STDC__on cc line */

#ifdef __STDC__
int
myfunc(const char *arg1, int arg2)
#else/* non-ANSI C case */
int
myfunc(arg1,arg2)
char *arg1,/* oops */
int arg2;
#endif
{
}
```

`unexpected "("`	
Type: Error	*Options:* All
A misplaced (was encountered in a `#if` or `#elif` directive.	

`#if 1 (`
`int i = 1;`
`#endif`	

`unexpected ")"`	
Type: Error	*Options:* All
A misplaced) was encountered in a `#if` or `#elif` directive.	

`#if ) 1`
`  int i = 1;`
`#endif`	

`unknown operand size:  op "`*operator*`"`	
Type: Error	*Options:* All
You applied *operator* `++`, `--`, or `=` to an operand whose size is unknown. The operand is usually a pointer to a structure or union whose members have not been declared.	

`f(void){`
`  struct s *sp;`
`  sp++;`
`}`	

unnamed *type* member	
Type: Warning	*Options:* All
In your *type* declaration, you failed to give a member a name.	
`union s { int; char c; };`	

unreachable case label: *value*	
Type: Warning	*Options:* All
The expression you specified in a `case` statement has a value outside the range of the type of the controlling expression of the enclosing switch statement. Therefore the case label can never be reached. In the message, *value* is represented as a hexadecimal value if the case expression is unsigned, decimal if it is signed.	

```
f(void){
    unsigned char uc;

    switch( uc ){
    case 256:
        ;
    }
}
```

unrecognized #pragma ignored: *pragma*	
Type: Warning	*Options:* `-v`
Because `#pragma` directives are implementation-specific, when the `-v` compilation flag is set, the C compiler warns about any such directives that it is ignoring. The C compiler does not recognize `#pragma` *pragma*.	
`#pragma list`	

`use "double" instead of "long float"`	
Type: Warning	*Options:* All
You declared an object or function to be `long float` , which was a synonym for `double`. ANSI C does not permit `long float` , although the C compiler accepts it as a transition aid.	
`long float f = 1.0;`	

`useless declaration`	
Type: Warning	*Options*: all
ANSI C requires that every declaration actually declare something, such as a declarator, a structure or union tag, enumeration constants. You wrote a declaration that provided no information to the compiler.	

`int; /* no identifier */`
`enum e { e1, e2 };/* introduces enum e */`
`enum e;/* no new information */`	

`using out of scope declaration:` *name*	
Type: Warning	*Options:* All
You previously declared *name* in a scope that is no longer active. In some ANSI C implementations, referring to such an object would yield an error; calling such a function would be interpreted as calling a function returning int. The C compiler remembers the previous declaration and uses it. This warning informs you what the compiler has done.	

```
f(void){
    extern int i;
    double sin(double);
}
g(void){
    double d = sin(1.5);
    i = 1;
}
```

`void expressions may not be arguments:` `arg` #*n*	
Type: Error	*Options:* All
A function call contains an argument for which the expression type is `void`.	

```
f(void){
    void v(void);
    g(v());
}
```

`void function cannot return value`	
Type: Warning	*Options:* All
You wrote a return statement with an expression, but the declared type of the function is `void`.	

`void v(void){`
`return 3;`
`}`	

`"void" must be sole parameter`	
Type: Error	*Options:* All
Only the first parameter in a function prototype declaration may have `void` type, and it must be the only parameter.	
`int f(int,void);`	

`void parameter cannot have name:` *name*	
Type: Error	*Options:* All
You have declared a parameter *name* in a function prototype declaration that has `void` type.	
`int f(void v);`	

zero or negative subscript	
Type: Warning, Error	*Options:* All
The size in an array declaration is zero or negative. The diagnostic is a warning if the size is zero and an error otherwise.	
`int ia[-5];` `int ib[0];`	

zero-sized struct/union	
Type: Error	*Options:* All
You declared a structure or union with size of zero.	
`struct s { int ia[0]; };`	

6.3 Operator Names

This section lists internal operator names that the compiler may use in error messages with definitions of these names.

`,OP`
The C "comma operator" (as distinct from the , that is used to separate function arguments).

`ARG`
A function argument. That is, a value passed to a function.

`AUTO`
An automatic variable that has not been allocated to a register.

`CALL`
A function call with arguments.

`CBRANCH`
A conditional branch. (This may be part of an `if` or loop statement.)

`CONV`

A conversion. It may have been explicit, in the form of a cast, or implicit, in the semantics of a C statement.

`FCON`
A floating-point constant.

`ICON`
An integer or address constant.

`NAME`
An object or function with `extern` or `static` storage class.

`PARAM`
A function parameter. That is, a value that is received by a function.

`REG`
An object that has been allocated to a register.

`RETURN`
The operation that corresponds to a `return` statement.

`STAR`
The indirection operator `*`, as in `*p`.

`STRING`
A string literal.

`U&`
The "take address of" operator (as distinct from the bit-wise AND operation).

`U-`
The arithmetic negation operator (as distinct from subtraction).

`UCALL`
A function call with no arguments.

`UGE`
An unsigned >= comparison.

`UGT`
An unsigned > comparison.

`ULE`
An unsigned <= comparison.

`ULT`
An unsigned < comparison.

`UPLUS`
The ANSI C "unary +" operator.

6.4 Other Error Messages

The following messages may appear at compile time, but they are not generated by the compiler. Messages beginning with `Assembler:` are produced by the assembler (`fbe`). Messages beginning with `ld:` are generated by `ld`, the link editor. Note that the format of the messages varies, and some of the messages are displayed over several lines.

`Assembler: file.c` `aline` *n* `(cline` *n*`) : trouble writing; probably out of temp-file space`
The file system may be low on space, or the temporary file or output file exceeded the current ulimit.

`Assembler: file.c aline` *n* (cline *n*) `Cannot open Output File` *filename*
The directory containing the source file is unwritable, or the file system containing source file is mounted read-only.

`ld: Symbol name in` *file2.o* is multiply defined. `First defined in` *file1.o*
A symbol name was defined more than once.

undefined symbol *sym1*	first referenced in file *file1.o*
ld fatal: Symbol referencing errors. No output written to a.out	
A referenced symbol was not found. Compilation terminates.	

Part 2—C Programming Tools

cscope *Source Code Browser*

Introduction

SourceBrowser, a window-oriented code browser that is more powerful than cscope, is described briefly on page 258. (SourceBrowser is sold separately.)

The cscope browser is an interactive program that locates specified elements of code in C, lex, or yacc source files. It lets you search and, if you want, edit your source files more efficiently than you could with a typical editor. That's because cscope knows about function calls — when a function is being called, when it is doing the calling — and C language identifiers and keywords. This chapter is a tutorial on the cscope browser, which is provided with this release.

How cscope *Works*

When cscope is called for a set of C, lex, or yacc source files, it builds a symbol cross-reference table for the functions, function calls, macros, variables, and preprocessor symbols in those files. It then lets you query that table about the locations of symbols you specify. First, it presents a menu and asks you to choose the type of search you would like to have performed. You may, for instance, want cscope to find all functions that call a specified function.

When cscope has completed this search, it prints a list. Each list entry contains the name of the file, the number of the line, and the text of the line in which cscope has found the specified code. In our case, the list will also include the names of the functions that call the specified function. You now have the option of requesting another search or examining one of the listed lines with the editor. If you choose the latter, cscope invokes the editor for

the file in which the line appears, with the cursor on that line. You may now view the code in context and, if you wish, edit the file as you would any other file. You can then return to the menu from the editor to request a new search.

Because the procedure you follow will depend on the task at hand, there is no single set of instructions for using `cscope`. For an extended example of its use, review the `cscope` session described in the next section. It shows how you can locate a bug in a program without learning all the code.

`cscope` — *Basic Use*

Suppose you are given responsibility for maintaining the program `prog`. You are told that an error message, `out of storage`, sometimes appears just as the program starts up. Now you want to use `cscope` to locate the parts of the code that are generating the message. Here is how you do it.

Step 1: Set Up the Environment

`cscope` is a screen-oriented tool that can only be used on terminals listed in the Terminal Information Utilities (`terminfo`) database. Be sure you have set the `TERM` environment variable to your terminal type so that `cscope` can verify that it is listed in the `terminfo` database. If you have not done so, assign a value to `TERM` and export it to the shell as follows:

Bourne Shell:

```
$ TERM=term_name; export TERM
```

C Shell:

```
% setenv TERM term_name
```

You may now want to assign a value to the `EDITOR` environment variable. By default, `cscope` invokes the `vi` editor. (The examples in this chapter illustrate `vi` usage.) If you prefer not to use `vi`, set the `EDITOR` environment variable to the editor of your choice and export `EDITOR`:

Bourne Shell:

```
$ EDITOR=emacs; export EDITOR
```

C Shell:

```
% setenv EDITOR emacs
```

Note that you may have to write an interface between `cscope` and your editor. For details, see "Command Line Syntax for Editors" on page 257.

If you want to use `cscope` only for browsing (without editing), you can set the `VIEWER` environment variable to `pg` and export `VIEWER`. `cscope` will then invoke `pg` instead of `vi`.

An environment variable called `VPATH` can be set to specify directories to be searched for source files. See "Using Viewpaths" on page 249.

Step 2: Invoke the `cscope` *Program*

By default, `cscope` builds a symbol cross-reference table for all the C, `lex`, and `yacc` source files in the current directory, and for any included header files in the current directory or the standard place. So if all the source files for the program to be browsed are in the current directory, and if its header files are there or in the standard place, invoke `cscope` without arguments:

```
$ cscope
```

To browse through selected source files, invoke `cscope` with the names of those files as arguments:

```
$ cscope file1.c file2.c file3.h
```

For other ways to invoke `cscope`, see "Command Line Options" on page 246.

`cscope` builds the symbol cross-reference table the first time it is used on the source files for the program to be browsed. By default, the table is stored in the file `cscope.out` in the current directory. On a subsequent invocation, `cscope` rebuilds the cross-reference only if a source file has been modified or the list of source files is different. When the cross-reference is rebuilt, the data for the unchanged files are copied from the old cross-reference, which makes rebuilding faster than the initial build and start-up time less for subsequent invocations.

Step 3: Locate the Code

Now let's return to the task we undertook at the beginning of this section: to identify the problem that is causing the error message `out of storage` to be printed. You have invoked `cscope`, the cross-reference table has been built. The `cscope` menu of tasks appears on the screen:

```
% cscope

cscope            Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Figure 7-1 The cscope Menu of Tasks

Press the `RETURN` key to move the cursor down the screen (with wraparound at the bottom of the display), and `^p` (control-p) to move the cursor up; or use the up (`(ua`) and down (`(da`) arrow keys if your keyboard has them. You can manipulate the menu, and perform other tasks, with the following single-key commands:

Table 7-1 `cscope` Menu Manipulation Commands (Sheet 1 of 2)

Menu Manipulation Commands	
`TAB`	move to next input field
`RETURN`	move to next input field
`^n`	move to next input field
`^p`	move to previous input field
`^y`	search with the last text typed
`^b`	move to previous input field and search pattern

Table 7-1 `cscope` Menu Manipulation Commands (Sheet 2 of 2)

Menu Manipulation Commands	
`^f`	move to next input field and search pattern
`^c`	toggle ignore/use letter case when searching (a search for `FILE` will match, for example, `file` and `File` when ignoring letter case)
`^r`	rebuild cross-reference
`!`	start an interactive shell (type `^d` to return to `cscope`)
`^l`	redraw the screen
`?`	display list of commands
`^d`	exit `cscope`

If the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a backslash (\) before the character.

Now move the cursor to the fifth menu item, `Find this text string`, enter the text `out of storage`, and press the `RETURN` key:

```
$ cscope

cscope           Press the ? key for help


Find this C symbol
Find this global definition
Find functions called by this function
Find functions calling this function
Find this text string:  out of storage
Change this text string
Find this egrep pattern
Find this file
Find files #including this file
```

Figure 7-2 Requesting a Search for a Text String

Note – Follow the same procedure to perform any other task listed in the menu except the sixth, `Change this text string`. Because this task is slightly more complex than the others, there is a different procedure for performing it. For a description of how to change a text string, see "Examples" on page 251.

`cscope` searches for the specified text, finds one line that contains it, and reports its finding as follows

```
Text string:  out of storage

  File Line
1 alloc.c 63 (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Figure 7-3 cscope Lists Lines Containing the Text String

After `cscope` shows you the results of a successful search, you have several options. You may want to change one of the lines or examine the code surrounding it in the editor. Or, if `cscope` has found so many lines that a list of them will not fit on the screen at once, you may want to look at the next part of the list. The following table shows the commands available after `cscope` has found the specified text:

Table 7-2 Commands for Use after an Initial Search (Sheet 1 of 2)

`1 - 9`	edit the file referenced by this line (the number you type corresponds to an item in the list of lines printed by `cscope`
`space`	display next set of matching lines
`+`	display next set of matching lines
`^v`	display next set of matching lines

Table 7-2 Commands for Use after an Initial Search (Sheet 2 of 2)

-	display previous set of matching lines
^e	edit displayed files in order
>	append the list of lines being displayed to a file
|	pipe all lines to a shell command

Again, if the first character of the text for which you are searching matches one of these commands, you can escape the command by entering a backslash before the character.

Now examine the code around the newly found line. Enter `1` (the number of the line in the list). The editor will be invoked with the file `alloc.c`; the cursor will be at the beginning of line 63 of `alloc.c`:

```
{
    return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char *p;
{
    if (p == NULL) {
        (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);
        exit(1);
    }
    return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

Figure 7-4 Examining a Line of Code Found by cscope

You can see that the error message is generated when the variable `p` is `NULL`. To determine how an argument passed to `alloctest()` could have been `NULL`, you must first identify the functions that call `alloctest()`.

Exit the editor by using normal quit conventions. You are returned to the menu of tasks. Now type `alloctest` after the fourth item, `Find functions calling this function:`

```
Text string:  out of storage

  File Line
1 alloc.c 63(void)fprintf(stderr,"\n%s:  out of storage\n",argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Figure 7-5 Requesting a List of Functions That Call alloctest()

cscope finds and lists three such functions:

```
Functions calling this function:  alloctest
File Function Line
1 alloc.c mymalloc 33 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc 43 return(alloctest(calloc((unsigned) nelem, (unsigned) size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned) size)));


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Figure 7-6 cscope Lists Functions That Call alloctest()

Now you want to know which functions call `mymalloc()`. `cscope` finds ten such functions. It lists nine of them on the screen and instructs you to press the space bar to see the rest of the list:

```
Functions calling this function:  mymalloc

    File          Function     Line
1   alloc.c       stralloc     24      return(strcpy(mymalloc(strlen(s) + 1), s));
2   crossref.c    crossref     47      symbol = (struct symbol *) mymalloc(msymbols *
                                                sizeof(struct symbol));
3   dir.c         makevpsrcdirs63      srcdirs = (char **) mymalloc(nsrcdirs * sizeof(char
                                                *));
4   dir.c         addincdir    167     incdirs = (char **) mymalloc(sizeof(char *));
5   dir.c         addincdir    168     incnames = (char **) mymalloc(sizeof(char *));
6   dir.c         addsrcfile   439     p = (struct listitem *) mymalloc(sizeof(struct
                                                listitem));
7   display.c     dispinit     87      displine = (int *) mymalloc(mdisprefs * sizeof(int ));
8   history.c     addcmd       19      h = (struct cmd *) mymalloc(sizeof(struct cmd));
9   main.c        main         212     s = mymalloc((unsigned ) (strlen(reffile) +
                                                strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Figure 7-7 `cscope` Lists Functions That Call mymalloc()

Because you know that the error message `out of storage` is generated at the beginning of the program, you can guess that the problem may have occurred in the function `dispinit()` (display initialization).

To view `dispinit()`, the seventh function on the list, type `7`:

```
void
dispinit()
{
            /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
        mdisprefs = 9;
    }
            /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char file[PATHLEN + 1]; /* file name */
    char function[PATLEN + 1];/* function name */
    char linenum[NUMLEN + 1]; /* line number */
    int screenline; /* screen line number */
    int width; /* source line display width */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

Figure 7-8 Viewing `dispinit()` in the Editor

`mymalloc()` failed because it was called either with a very large number or a negative number. By examining the possible values of `FLDLINE` and `REFLINE`, you can see that there are situations in which the value of `mdisprefs` is negative, that is, in which you are trying to call `mymalloc()` with a negative number.

Step 4: Edit the Code

On a windowing terminal you may have multiple windows of arbitrary size. The error message `out of storage` might have appeared as a result of running `prog` in a window with too few lines. In other words, that may have been one of the situations in which `mymalloc()` was called with a negative

number. Now you want to be sure that when the program aborts in this situation in the future, it does so after printing the more meaningful error message `screen too small`. Edit the function `dispinit()` as follows:

```
/* initialize display parameters */
void
dispinit()
{
            /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs <= 0) {
        (void) fprintf(stderr,"\n%s:  screen too small\n", argv0);
        exit(1);
    }
    if (mdisprefs > 9)
        mdisprefs = 9;
            /* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
```

Figure 7-9 Using `cscope` to Fix the Problem

You have fixed the problem we began investigating at the beginning of this section. Now if `prog` is run in a window with too few lines, it will not simply fail with the unedifying error message `out of storage`. Instead, it will check the window size and generate a more meaningful error message before exiting.

Command Line Options

As noted, `cscope` builds a symbol cross-reference table for the C, `lex`, and source files in the current directory by default. That is,

```
$ cscope
```

is equivalent to

```
$ cscope *.[chly]
```

We have also seen that you can browse through selected source files by invoking `cscope` with the names of those files as arguments:

```
$ cscope file1.c file2.c file3.h
```

`cscope` provides command line options that allow you greater flexibility in specifying source files to be included in the cross-reference. When you invoke `cscope` with the `-s` option and any number of directory names (separated by commas)

```
$ cscope -s dir,dir,dir
```

`cscope` will build a cross-reference for all the source files in the specified directories as well as the current directory. To browse through all of the source files whose names are listed in *file* (file names separated by spaces, tabs, or new-lines), invoke `cscope` with the `-i` option and the name of the file containing the list:

```
$ cscope -i file
```

If your source files are in a directory tree, the following commands will allow you to browse through all of them easily:

```
$ find .  -name '*.[chly] ' -print | sort > file
$ cscope -i file
```

Note that if this option is selected, `cscope` ignores any other files appearing on the command line.

The `-I` option to `cscope` is similar to the `-I` option to `cc`. By default, `cscope` searches for included header files in the current directory, then the standard place. If you want `cscope` to search for an included header file in a different directory, specify the path of the directory with `-I`:

```
$ cscope -I dir
```

In this case, `cscope` will search the directory *dir* for `#include` files called into the source files in the current directory. Directories are searched for `#include` files in the following order:

1. the current directory
2. the directories specified with `-I`
3. the standard place for header files (usually `/usr/include`)

You can invoke the `-I` option more than once on a command line. `cscope` will search the specified directories in the order they appear on the command line.

You can specify a cross-reference file other than the default `cscope.out` by invoking the `-f` option. This is useful for keeping separate symbol cross-reference files in the same directory. You may want to do this if two programs are in the same directory, but do not share all the same files:

```
$ cscope -f admin.ref admin.c common.c aux.c libs.c
$ cscope -f delta.ref delta.c common.c aux.c libs.c
```

In this example, the source files for two programs, `admin` and `delta`, are in the same directory, but the programs consist of different groups of files. By specifying different symbol cross-reference files when you invoke `cscope` for each set of source files, the cross-reference information for the two programs is kept separate.

You can use the `-p`*n* option to specify that `cscope` display the path name, or part of the path name, of a file when it lists the results of a search. The number you give to `-p` stands for the last *n* elements of the path name you want to be displayed. The default is `1`, the name of the file itself. So if your current directory is `home/common`, the command

```
$ cscope -p2
```

will cause `cscope` to display `common/file1.c`, `common/file2.c`, and so forth when it lists the results of a search.

If the program you want to browse contains a large number of source files, you can use the -b option to tell cscope to stop after it has built a cross-reference; cscope will not display a menu of tasks. When you use cscope -b in a pipeline with the batch(1) command (described in the *SunOS 5.0 Reference Manual*) cscope will build the cross-reference in the background:

```
$ echo 'cscope -b' | batch
```

Once the cross-reference is built (and as long as you have not changed a source file or the list of source files in the meantime), you need only specify

```
$ cscope
```

for the cross-reference to be copied and the menu of tasks to be displayed in the normal way. In other words, you can use this sequence of commands when you want to continue working without having to wait for cscope to finish its initial processing.

The -d option instructs cscope not to update the symbol cross-reference. You can use it to save time — cscope will not check the source files for changes — if you are sure that no such changes have been made.

Note – Use the -d option with care. If you specify -d under the erroneous impression that your source files have not been changed, cscope will refer to an outdated symbol cross-reference in responding to your queries.

Check the cscope(1) page in the *SunOS 5.0 Reference Manual* for other command line options.

Using Viewpaths

As we have seen, cscope searches for source files in the current directory by default. When the environment variable VPATH is set, cscope searches for source files in directories that comprise your viewpath. A viewpath is an ordered list of directories, each of which has the same directory structure below it.

For example, suppose you are part of a software project. There is an *official* set of source files in directories below /fs1/ofc. Each user has a home directory (/usr/you). If you make changes to the software system, you may have

copies of just those files you are changing in `/usr/you/src/cmd/prog1`. The official versions of the entire program can be found in the directory `/fs1/ofc/src/cmd/prog1`.

Suppose you use `cscope` to browse through the three files that comprise `prog1`, namely, `f1.c`, `f2.c`, and `f3.c`. You would set `VPATH` to `/usr/you` and `/fs1/ofc`

and export it, as in

Bourne Shell:

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

C Shell:

```
% setenv VPATH /usr/you:/fs1/ofc
```

You would then make your current directory `/usr/you/src/cmd/prog1`, and invoke `cscope`:

```
$ cscope
```

The program will locate all files in the viewpath. In case duplicates are found, `cscope` uses the file whose parent directory appears earlier in `VPATH`. Thus, if `f2.c` is in your directory (and all three files are in the official directory), `cscope` will examine `f2.c` from your directory and `f1.c` and `f3.c` from the official directory.

The first directory in `VPATH` must be a prefix (usually `$HOME`) of the directory you will be working in. Each colon-separated directory in `VPATH` must be absolute: it should begin at `/`.

Stacking `cscope` *and Editor Calls*

`cscope` and editor calls can be stacked. That means that when `cscope` puts you in the editor to view a reference to a symbol and there is another reference of interest, you can invoke `cscope` again from within the editor to view the second reference without exiting the current invocation of either `cscope` or the editor. You can then back up by exiting the most recent invocation with the appropriate `cscope` and editor commands.

Examples

This section presents examples of how `cscope` can be used to perform three tasks: changing a constant to a preprocessor symbol, adding an argument to a function, and changing the value of a variable. The first example demonstrates the procedure for changing a text string, which differs slightly from the other tasks on the `cscope` menu. That is, once you have entered the text string to be changed, `cscope` prompts you for the new text, displays the lines containing the old text, and waits for you to specify which of these lines you want it to change.

Changing a Constant to a Preprocessor Symbol

Suppose you want to change a constant, 100, to a preprocessor symbol, `MAXSIZE`. Select the sixth menu item, `Change this text string`, and enter `\100`. The `1` must be escaped with a backslash because it has a special meaning (item 1 on the menu) to `cscope`. Now press `RETURN`. `cscope` will prompt you for the new text string. Type `MAXSIZE`:

```
cscope     Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

Figure 7-10 Changing a Text String

cscope displays the lines containing the specified text string, and waits for you to select those in which you want the text to be changed:

```
Change "100" to "MAXSIZE"

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;    /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

Figure 7-11 cscope Prompts for Lines to Changed

You know that the constant 100 in lines 1, 2, and 3 of the list (lines 4, 26, and 8 of the listed source files) should be changed to MAXSIZE. You also know that 0100 in read.c and 100.0 in err.c (lines 4 and 5 of the list) should not be changed. You select the lines you want changed with the following single-key commands:

Table 7-3 Commands for Selecting Lines to Be Changed (Sheet 1 of 2)

1-9	mark or unmark the line to be changed
*	mark or unmark all displayed lines to be changed
space	display next set of lines
+	display next set of lines

Table 7-3 Commands for Selecting Lines to Be Changed (Sheet 2 of 2)

-	display previous set of lines
a	mark all lines to be changed
^d	change the marked lines and exit
ESC	exit without changing the marked lines

In this case, enter `1`, `2`, and `3`. Note that the numbers you type are not printed on the screen. Instead, `cscope` marks each list item you want to be changed by printing a > (greater than) symbol after its line number in the list:

```
Change "100" to "MAXSIZE"

  File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
Select lines to change (press the ? key for help):
```

Figure 7-12 Marking Lines to Be Changed

Now type ^d to change the selected lines. cscope displays the lines that have been changed and prompts you to continue:

```
Changed lines:

    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {

Press the RETURN key to continue:
```

Figure 7-13 cscope Displays Changed Lines of Text

When you press RETURN in response to this prompt, cscope redraws the screen, restoring it to its state before you selected the lines to be changed, as shown in Figure 7-17.

The next step is to add the `#define` for the new symbol `MAXSIZE`. Because the header file in which the `#define` is to appear is not among the files whose lines are displayed, you must escape to the shell by typing `!`. The shell prompt will appear at the bottom of the screen. Then enter the editor and add the `#define`:

```
Text string:  100

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;        /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

Figure 7-14 Escaping from `cscope` to the Shell

To resume the `cscope` session, quit the editor and type `^d` to exit the shell.

Adding an Argument to a Function

Adding an argument to a function involves two steps: editing the function itself and adding the new argument to every place in the code where the function is called. `cscope` makes that easy.

First, edit the function by using the second menu item, `Find this global definition`. Next, find out where the function is called. Use the fourth menu item, `Find functions calling this function`, to get a list of all the functions that call it. With this list, you can either invoke the editor for each line found by entering the list number of the line individually, or invoke the

editor for all the lines automatically by typing `^e`. Using `cscope` to make this kind of change assures that none of the functions you need to edit will be overlooked.

Changing the Value of a Variable

The value of `cscope` as a browser becomes apparent when you want to see how a proposed change will affect your code. Suppose you want to change the value of a variable or preprocessor symbol. Before doing so, use the first menu item, `Find this C symbol`, to obtain a list of references that will be affected. Then use the editor to examine each one. This will help you predict the overall effects of your proposed change. Later, you can use `cscope` in the same way to verify that your changes have been made.

Notes

This section describes certain problems that may arise when you use `cscope` and how to avoid them.

Unknown Terminal Type

You may see the error message:

```
Sorry, I don't know how to deal with your "term" terminal
```

If this message appears, your terminal may not be listed in the Terminal Information Utilities (`terminfo`) database that is currently loaded. Make sure you have assigned the correct value to `TERM`. If the message reappears, try reloading the Terminal Information Utilities. You may also see:

```
Sorry, I need to know a more specific terminal type than "unknown"
```

If this message appears, set and export the `TERM` variable as described in "Step 1: Set Up the Environment" on page 236.

Command Line Syntax for Editors

As noted, `cscope` invokes the `vi` editor by default. You may override the default setting by assigning your preferred editor to the `EDITOR` environment variable and exporting `EDITOR`, as described in "Step 1: Set Up the Environment" on page 236. Note, however, that `cscope` expects the editor it uses to have a command line syntax of the form

```
$ editor +linenum filename
```

as does `vi`. If the editor you want to use does not have this command line syntax, you must write an interface between `cscope` and the editor.

Suppose you want to use `ed`, for example. Because `ed` does not allow specification of a line number on the command line, you will not be able to use it to view or edit files with `cscope` unless you write a shell script (called `myedit` here) that contains the following line:

```
/usr/bin/ed $2
```

Now set the value of `EDITOR` to your shell script and export `EDITOR`:

Bourne Shell:

```
$ EDITOR=myedit; export EDITOR
```

C Shell:

```
% setenv EDITOR myedit
```

When `cscope` invokes the editor for the list item you have specified, say, line 17 in `main.c`, it will invoke your shell script with the command line

```
$ myedit +17 main.c
```

`myedit` will discard the line number (`$1`) and call `ed` correctly with the file name (`$2`). Of course, you will then have to execute the appropriate `ed` commands to display and edit the line. That is, you will not be moved automatically to line 17 of the file.

SourceBrowser

SourceBrowser is an interactive tool to aid programmers in the development and maintenance of software systems, particularly large ones. Because SourceBrowser builds a database and uses it to respond to queries, once the database it built, the size of the code you are browsing has minimal impact on SourceBrowser's speed.

SourceBrowser can help you find *all* occurences of any symbol of your choice, including those found in header files. It can be used from either a command-line or window environment.

SourceBrowser uses a *what you see is what you browse* paradigm. The source code you manipulate is the same source code SourceBrowser uses in its searches. This allows you to edit code from within SourceBrowser.

SourceBrowser is designed to be used with multiple languages. In addition to C, it can be used with FORTRAN , C++, Pascal and Modula-2.

SourceBrowser is sold separately. For more information, see the *Browsing Source Code* manual.

`lint` Source Code Checker

Scope of this Chapter

Note – Of the nearly five hundred diagnostics issued by `lint`, *this chapter describes only the much smaller subset of* `lint` *–specific warnings: those not also issued by the compiler.* The one exception to this rule applies to diagnostics issued both by `lint` and the compiler that are capable of being suppressed only by `lint` options.

For the text and examples of messages issued *exclusively by* `lint` or *subject exclusively to its options,* refer to "lint-specific Messages" on page 274.

For the messages *also issued by the compiler,* consult the *C 2.0 Programmer's Guide.*

Introduction

`lint` checks for code constructs that may cause your C program not to compile, or to execute with unexpected results. `lint` issues every error and warning message produced by the C compiler. It also issues `lint` –specific warnings about potential bugs and portability problems.

In particular, `lint` compensates for separate and independent compilation in C by flagging inconsistencies in definition and use across files, including any libraries you have used. In a large project environment especially, where the same function may be used by different programmers in hundreds of separate

modules of code, `lint` can help discover bugs that otherwise might be difficult to find. A function called with one less argument than expected, for example, looks at the stack for a value the call has never pushed, with results correct in one condition, incorrect in another, depending on whatever happens to be in memory at that stack location. By identifying dependencies like this one, and dependencies on machine architecture as well, `lint` can improve the reliability of code run on your machine or someone else's.

Options and Directives

`lint` is a static analyzer, which means that it cannot evaluate the run-time consequences of the dependencies it detects. Certain programs, for instance, may contain hundreds of unreachable `break` statements, of little importance, about which you typically can do nothing, and which `lint` will faithfully flag nevertheless. That's where `lint`'s command line options and directives — special comments embedded in the source text — come in. For the example we've cited here,

- you can invoke `lint` with the `-b` option to suppress all complaints about unreachable `break` statements;
- for a finer-grained control, you can precede any unreachable statement with the comment `/* NOTREACHED */` to suppress the diagnostic for that statement.

"Usage" on page 265 discusses options and directives in greater detail and introduces the `lint` filter technique, which lets you tailor `lint`'s behavior even more finely to your project's needs. It also shows you how to use `lint` libraries to check your program for compatibility with the library functions you have called in it.

Message Formats

Most of `lint`'s messages are simple, one-line statements printed for each occurrence of the problem they diagnose. Errors detected in included files are reported multiply by the compiler but only once by `lint`, no matter how many times the file is included in other source files. Compound messages are issued for inconsistencies across files and, in a few cases, for problems within them as well. A single message describes every occurrence of the problem in the file or files being checked. When use of a `lint` filter (see "Usage" on page 265) requires that a message be printed for each occurrence, compound diagnostics can be converted to the simple type by invoking `lint` with the `-s` option.

What `lint` *Does*

`lint` –specific diagnostics are issued for three broad categories of conditions: inconsistent use, nonportable code, and suspicious constructs. In this section, we'll review examples of `lint`'s behavior in each of these areas, and suggest possible responses to the issues they raise.

Consistency Checks

Inconsistent use of variables, arguments, and functions is checked within files as well as across them. Generally speaking, the same checks are performed for prototype uses, declarations, and parameters as `lint` (1) checks for for old-style functions. (If your program does not use function prototypes, `lint` will check the number and types of parameters in each call to a function more strictly than the compiler.) `lint` also identifies mismatches of conversion specifications and arguments in `[fs]printf()` and `[fs]scanf()` control strings. Examples:

- Within files, `lint` flags non `void` functions that *fall off the bottom* without returning a value to the invoking function. In the past, programmers often indicated that a function was not meant to return a value by omitting the return type: `fun() {}`. That convention means nothing to the compiler, which regards `fun()` as having the return type `int`. Declare the function with the return type `void` to eliminate the problem.
- Across files, `lint` detects cases where a non`void` function does not return a value, yet is used for its value in an expression, and the opposite problem, a function returning a value that is sometimes or always ignored in subsequent calls. When the value is always ignored, it may indicate an inefficiency in the function definition. When it is sometimes ignored, it's probably bad style (typically, not testing for error conditions). If you do not need to check the return values of string functions like `strcat()`, `strcpy()`, and `sprintf()`, or output functions like `printf()` and `putchar()`, cast the offending call(s) to `void`.
- `lint` identifies variables or functions that are declared but not used or defined; used but not defined; or defined but not used. That means that when `lint` is applied to some, but not all files of a collection to be loaded together, it will complain about functions and variables declared in those files but defined or used elsewhere; used there but defined elsewhere; or defined there and used elsewhere. Invoke the `-x` option to suppress the former complaint, `-u` to suppress the latter two.

Portability Checks

Some nonportable code is flagged by `lint` in its default behavior, and a few more cases are diagnosed when `lint` is invoked with `-p` and/or `-Xc`. The latter tells `lint` to check for constructs that do not conform to the ANSI C standard. For the messages issued under `-p` and `-Xc`, check the "Usage" section below. Examples:

- In some C language implementations, character variables that are not explicitly declared `signed` or `unsigned` are treated as signed quantities with a range typically from –128 to 127. In other implementations, they are treated as nonnegative quantities with a range typically from 0 to 255. So the test

```
char c;
c = getchar();
if (c == EOF) ...
```

where `EOF` has the value –1, will always fail on machines where character variables take on nonnegative values. One of `lint`'s `-p` checks will flag any comparison that implies a *plain* `char` may have a negative value. Note, however, that declaring `c` a `signed char` in the above example eliminates the diagnostic, not the problem. That's because `getchar()` must return all possible characters and a distinct `EOF` value, so a `char` cannot store its value. We cite this example, perhaps the most common one arising from implementation-defined sign-extension, to show how a thoughtful application of `lint`'s portability option can help you discover bugs not related to portability. In any case, declare `c` as an `int`.

- A similar issue arises with bit-fields. When constant values are assigned to bit-fields, the field may be too small to hold the value. On a machine that treats bit-fields of type `int` as unsigned quantities, the values allowed for `int x:3` range from 0 to 7, whereas on machines that treat them as signed quantities, they range from –4 to 3. However unintuitive it may seem, a three-bit field declared type `int` cannot hold the value 4 on the latter machines. `lint` invoked with `-p` flags all bit-field types other than `unsigned int` or `signed int`. Note that these are the only *portable* bit-field types. Sun C supports `int`, `char`, `short`, and `long` bit-field types that may be `unsigned`, `signed`, or *plain*. It also supports the `enum` bit-field type.

- Bugs can arise when a larger-sized type is assigned to a smaller-sized type. If significant bits are truncated, accuracy is lost:

```
short s;
long l;
s = l;
```

`lint` flags all such assignments by default; the diagnostic can be suppressed by invoking the `-a` option. Bear in mind that you may be suppressing other diagnostics when you invoke `lint` with this or any other option. Check the list in the "Usage" section below for the options that suppress more than one diagnostic.

- A cast of a pointer to one object type to a pointer to an object type with stricter alignment requirements may not be portable. `lint` flags

```
int *fun(y)
char *y;
{
    return(int *)y;
}
```

because, on most machines, an `int` cannot start on an arbitrary byte boundary, whereas a `char` can. You can suppress the diagnostic by invoking `lint` with `-h`, although, again, you may be disabling other messages. Better still, eliminate the problem by using the generic pointer `void *`.

- ANSI C leaves the order of evaluation of complicated expressions undefined. What this means is that when function calls, nested assignment statements, or the increment and decrement operators cause side effects — when a variable is changed as a by-product of the evaluation of an expression — the order in which the side effects take place is highly machine dependent. By default, `lint` flags any variable changed by a side effect and used elsewhere in the same expression:

```
int a[10];
main()
{
    int i = 1;
    a[i++] = i;
}
```

Note that in this example the value of `a[1]` may be 1 if one compiler is used, 2 if another. The bitwise logical operator `&` can give rise to this diagnostic when it is mistakenly used in place of the logical operator `&&`:

```
if ((c = getchar()) != EOF & c != '0')
```

Suspicious Constructs

`lint` flags a miscellany of legal constructs that may not represent what the programmer intended. Examples:

- An `unsigned` variable always has a nonnegative value. So the test

```
unsigned x;
if (x < 0) ...
```

will always fail. Whereas the test

```
unsigned x;
if (x > 0) ...
```

is equivalent to

```
if (x != 0) ...
```

which may not be the intended action. `lint` flags suspicious comparisons of `unsigned` variables with negative constants or 0. To compare an `unsigned` variable to the bit pattern of a negative number, cast it to `unsigned`:

```
if (u == (unsigned) -1) ...
```

Or use the `U` suffix:

```
if (u == -1U) ...
```

- `lint` flags expressions without side effects that are used in a context where side effects are expected, that is, where the expression may not represent what the programmer intended. It issues an additional warning whenever the equality operator is found where the assignment operator was expected, in other words, where a side effect was expected:

```
int fun()
{
    int a, b, x, y;
    (a = x) && (b == y);
}
```

- `lint` cautions you to parenthesize expressions that mix both the logical and bitwise operators (specifically, `&` , `|` (, `^` , `<<` , `>>`), where misunderstanding of operator precedence may lead to incorrect results. Because the precedence of bitwise `&` , for example, falls below logical `= =` , the expression

```
if (x & a == 0) ...
```

will be evaluated as

```
if (x & (a == 0)) ...
```

which is most likely not what you intended. Invoking `lint` with `-h` disables the diagnostic.

Usage

You invoke `lint` with a command of the form

```
$ lint file.c file.c
```

`lint` examines code in two passes. In the first, it checks for error conditions local to C source files, in the second for inconsistencies across them. This process is invisible to the user unless `lint` is invoked with `-c`:

```
$ lint -c file1.c file2.c
```

That command directs `lint` to execute the first pass only and collect information relevant to the second — about inconsistencies in definition and use across `file1.c` and `file2.c` — in intermediate files named `file1.ln` and `file2.ln`:

```
$ ls
file1.c
file1.ln
file2.c
file2.ln
```

In this way, the `-c` option to `lint` is analogous to the `-c` option to `cc`, which suppresses the link editing phase of compilation. Generally speaking, `lint` 's command line syntax closely follows `cc`'s.

When the `.ln` files are `lint`ed

```
$ lint file1.ln file2.ln
```

the second pass is executed. `lint` processes any number of `.c` or `.ln` files in their command line order. So

```
$ lint file1.ln file2.ln file3.c
```

directs `lint` to check `file3.c` for errors internal to it and all three files for consistency.

`lint` searches directories for included header files in the same order as `cc` (see the *C 2.0 Programmer's Guide*). You can use the `-I` option to `lint` as you would the `-I` option to `cc`. Namely, if you want `lint` to check an included header file that is stored in a directory other than your current directory or the standard place, specify the path of the directory with `-I` as follows:

```
$ lint -Idir file1.c file2.c
```

You can specify -I more than once on the lint command line. Directories are searched in the order they appear on the command line. Of course, you can specify multiple options to lint on the same command line. Options may be concatenated unless one of the options takes an argument:

```
$ lint -cp -Idir -Idir file1.c file2.c
```

That command directs lint to

- execute the first pass only;
- perform additional portability checks;
- search the specified directories for included header files.

lint *Libraries*

You can use lint libraries to check your program for compatibility with the library functions you have called in it: the declaration of the function return type, the number and types of arguments the function expects, and so on. The standard lint libraries correspond to libraries supplied by the C compilation system, and generally are stored in the standard place on your system, the directory /usr/ccs/lib. By convention, lint libraries have names of the form llib-lx.ln.

The lint standard C library, llib-lc.ln, is appended to the lint command line by default; checks for compatibility with it can be suppressed by invoking the -n option. Other lint libraries are accessed as arguments to -l. That is,

```
$ lint -lx file1.c file2.c
```

directs lint to check the usage of functions and variables in file1.c and file2.c for compatibility with the lint library llib-lx.ln. The library file, which consists only of definitions, is processed exactly as are ordinary source files and ordinary .ln files, except that functions and variables used inconsistently in the library file, or defined in the library file but not used in the source files, elicit no complaints.

To create your own `lint` library, insert the directive `/* LINTLIBRARY */` at the head of a C source file, then invoke `lint` for that file with the `-o` option and the library name that will be given to `-l`:

```
$ lint -o files headed by /* LINTLIBRARY */
```

causes only definitions in the source files headed by `/* LINTLIBRARY */` to be written to the file `llib-lx.ln`. (Note the analogy of `lint -o` to `cc -o`.) A library can be created from a file of function prototype declarations in the same way, except that both `/* LINTLIBRARY */` and `/* PROTOLIB`*n*`*/` must be inserted at the head of the declarations file. If *n* is 1, prototype declarations will be written to a library `.ln` file just as are old-style definitions. If *n* is 0, the default, the process is cancelled. Invoking `lint` with `-y` is another way of creating a `lint` library:

```
$ lint -y -ox file1.c file2.c
```

causes each source file named on the command line to be treated as if

it began with `/* LINTLIBRARY */` and only its definitions to be written to `llib-lx.ln`.

By default, `lint` searches for `lint` libraries in the standard place. To direct `lint` to search for a `lint` library in a directory other than the standard place, specify the path of the directory with the `-L` option:

```
$ lint -Ldir -lx file1.c file2.c
```

The specified directory is searched before the standard place.

`lint` *Filters*

A `lint` filter is a project-specific post-processor that typically uses an `awk` script or similar program to read the output of `lint` and discard messages that your project has decided do not identify real problems — string functions, for instance, returning values that are sometimes or always ignored. It enables you to generate customized diagnostic reports when `lint` options and directives do not provide sufficient control over output.

Two options to `lint` are particularly useful in developing a filter. Invoking `lint` with `-s` causes compound diagnostics to be converted into simple, one-line messages issued for each occurrence of the problem diagnosed. The easily parsed message format is suitable for analysis by an `awk` script.

Invoking `lint` with `-k` causes certain comments you have written in the source file to be printed in output, and can be useful both in documenting project decisions and specifying the post-processor's behavior. In the latter instance, if the comment identified an expected `lint` message, and the reported message was the same, the message might be filtered out. To use `-k`, insert on the line preceding the code you wish to comment the `/* LINTED [msg] */` directive, where *msg* refers to the comment to be printed when `lint` is invoked with `-k`. (Refer to the list of directives below for what `lint` does when `-k` is *not* invoked for a file containing `/* LINTED [msg] */`.)

Options and Directives Listed

These options suppress specific messages:

Table 8-1 lint Options Supressing Messages (Sheet 1 of 2)

Option	Suppresses
-a	assignment causes implicit narrowing conversion
	conversion to larger integral type may sign-extend incorrectly
-b	statement not reached (unreachable break and empty statements)
-h	assignment operator "=" found where equality operator "= =" was expected
	constant operand to op: "!"
	fallthrough on case statement
	pointer cast may result in improper alignment
	precedence confusion possible; parenthesize
	statement has no consequent: if
	statement has no consequent: else
-m	declared global, could be static

Table 8-1 lint Options Supressing Messages (Sheet 2 of 2)

Option	Suppresses
-u	name defined but never used
	name used but not defined
-v	argument unused in function
-x	name declared but never used or defined

These options enable specific messages:

Table 8-2 lint Options Enabling Messages

Option	Enables
-p	conversion to larger integral type may sign-extend incorrectly
	may be indistinguishable due to truncation or case
	pointer casts may be troublesome
	nonportable bit-field type
	suspicious comparison of char with *value*: op "*op*"
-Xc	bitwise operation on signed value nonportable
	function must return int: main()
	may be indistinguishable due to truncation or case
	only 0 or 2 parameters allowed: main()
	nonportable character constant

Other options:

-C *filename*
Create a `.ln` file with the filename specified. These `.ln` files are the product of `lint`'s first pass only. *filename* may be a complete pathname.

-c
Create a `.ln` file consisting of information relevant to `lint`'s second pass for every `.c` file named on the command line. The second pass is not executed.

`-F`
When referring to the `.c` files named on the command line, print their path names as supplied on the command line rather than only their base names.

`-I`*dir*
Search the directory *dir* for included header files.

`-k`
When used with the directive `/* LINTED [`*msg*`] */`, print `info: `*msg*.

`-l`*x*
Access the `lint` library `llib-lx.ln`.

`-L`*dir*
When used with `-l`, search for a `lint` library in the directory *dir*.

`-n`
Suppress checks for compatibility with the default `lint` standard C library.

`-o`*x*
Create the file `llib-lx.ln`, consisting of information relevant to `lint`'s second pass, from the `.c` files named on the command line. Generally used with `-y` or `/* LINTLIBRARY */` to create `lint` libraries.

`-s`
Convert compound messages into simple ones.

`-y`
Treat every `.c` file named on the command line as if it began with the directive `/* LINTLIBRARY */`.

`-V`
Write the product name and release to standard error.

Directives:

Table 8-3 lint Directives

Directive	Action
/*ARGSUSED*n*/	Suppress: `argument unused in function` for every argument but the first *n* in the function definition it precedes. Default is 0.

/*CONSTCOND*/	Suppress:
	`constant in conditional context`
	`constant operand to op: "!"`
	`logical expression always false: op "&&"`
	`logical expression always true: op "||"`
	for the constructs it precedes. Also
	`/* CONSTANTCONDITION */`.

/*EMPTY*/	Suppress:
	`statement has no consequent: else`
	`when inserted between the else and semicolon;`
	`statement has no consequent: if`
	when inserted between the controlling expression of the `if` and semicolon.

/*FALLTHRU*/	Suppress: `fallthrough on case statement` for the `case` statement it precedes. Also `/* FALLTHROUGH */`.

Table 8-3 lint Directives

Directive	Action

/*LINTED~[*msg*]*/	When -k is not invoked, suppress every warning pertaining to an intrafile problem except:
	`argument unused in function`
	`declaration unused in block`
	`set but not used in function`
	`static unused`
	`variable unused in function`
	for the line of code it precedes. *msg* is ignored.

/*LINTLIBRARY*/	When -o is invoked, write to a library .ln file only definitions in the .c file it heads.

/*NOTREACHED*/	Suppress:
	`statement not reached`
	for the unreached statements it precedes;
	`fallthrough on case statement`
	for the case it precedes that cannot be reached from the preceding case;
	`function falls off bottom without returning value`
	for the closing curly brace it precedes at the end of the function.

Table 8-3 lint Directives

Directive	Action

/*PRINTFLIKE*n**/	Treat the *n*th argument of the function definition it precedes as a `[fs]printf()` format string and issue:
	`malformed format string`
	for invalid conversion specifications in that argument, and
	`function argument type inconsistent with format`
	`too few arguments for format`
	`too many arguments for format`
	for mismatches between the remaining arguments and the conversion specifications. `lint` issues these warnings by default for errors in calls to `[fs]printf()` functions provided by the standard C library.

/*PROTOLIB*n**/	When *n* is 1 and `/* LINTLIBRARY */` is used, write to a library `.ln` file only function prototype declarations in the `.c` file it heads. Default is 0, cancelling the process.
/*SCANFLIKE*n**/	Same as `/* PRINTFLIKE`*n*` */` except that the *n*th argument of the function definition is treated as a `[fs]scanf()` format string. By default, `lint` issues warnings for errors in calls to `[fs]scanf()` functions provided by the standard C library.
/*VARARGS*n**/	For the function whose definition it precedes, suppress: `function called with variable number of arguments` for calls to the function with *n* or more arguments.

`lint`-*specific Messages*

This section lists alphabetically the warning messages issued *exclusively by* `lint` or *subject exclusively to its options*. The code examples illustrate conditions in which the messages are elicited. Note that some of the examples would elicit messages in addition to the one stated. For the remaining `lint` messages, consult the *C 2.0 Programmer's Guide*.

Format is explained on page 260.

`argument unused in function`

Format: Compound

A function argument was not used. Preceding the function definition with `/* ARGSUSEDn */` suppresses the message for all but the first *n* arguments; invoking lint with `-v` suppresses it for every argument.

```
1 int fun(int x, int y)
2 {
3   return x;
4 }
5 /* ARGSUSED1 */
6 int fun2(int x, int y)
7 {
8   return x;
9 }
============
argument unused in function
    (1) y in fun
```

`array subscript cannot be >`*value*`:` *value*

Format: Simple

The value of an array element's subscript exceeded the upper array bound.

```
1 int fun()
2 {
3   int a[10];
4   int *p = a;
5   while (p != &a[10]) /* using address is ok */
6       p++;
7   return a[5 + 6];
8 }
============
(7) warning: array subscript cannot be > 9: 11
```

`array subscript cannot be negative:` *value*
Format: Simple
The constant expression that represents the subscript of a true array (as opposed to a pointer) had a negative value.

`1 int f()`
`2 {`
`3  int a[10];`
`4  return a[5 * 2 / 10 - 2];`
`5 }`
`============`
`(4) warning: array subscript cannot be negative: -1`

`assignment causes implicit narrowing conversion`
Format: Compound
An object was assigned to one of a smaller type. Invoking lint with `-a` suppresses the message. So does an explicit cast to the smaller type.

`1 void fun()`
`2 {`
`3  short s;`
`4  long l = 0;`
`5  s = l;`
`6 }`
`============`
`assignment causes implicit narrowing conversion`
`  (5)`

`assignment operator = found where == was expected`

Format: Simple

An assignment operator was found where a conditional expression was expected. The message is not issued when an assignment is made to a variable using the value of a function call or in the case of string copying (see the example below). The warning is suppressed when lint is invoked with `-h`.

```
1 void fun()
2 {
3   char *p, *q;
4   int a = 0, b = 0, c = 0, d = 0, i;
5   i = (a = b) && (c == d);
6   i = (c == d) && (a = b);
7   if (a = b)
8       i = 1;
9   while (*p++ = *q++);
10  while (a = b);
11  while ((a = getchar()) == b);
12 if (a = foo()) return;
13 }
============
(5) warning: assignment operator "=" found where "=="
was expected
(7) warning: assignment operator "=" found where "=="
was expected
(10) warning: assignment operator "=" found where "=="
was expected
```

`bitwise operation on signed value nonportable`

Format: Compound

The operand of a bitwise operator was a variable of signed integral type, as defined by ANSI C. Because these operators return values that depend on the internal representations of integers, their behavior is implementation-defined for operands of that type. The message is issued only when lint is invoked with `-Xc`.

```
1 fun()
2 {
3   int i;
4   signed int j;
5   unsigned int k;
6   i = i & 055;
7   j = j | 022;
8   k = k >> 4;
9 }
============
warning: bitwise operation on signed value nonportable
(6) (7)
```

`constant in conditional context`

Format: Simple

The controlling expression of an if, while, or for statement was a constant. Preceding the statement with `/* CONSTCOND */` suppresses the message.

```
1 void fun()
2 {
3   if (! 1) return;
4   while (1) foo();
5   for (;1;);
6   for (;;);
7   /* CONSTCOND */
8   while (1);
9 }
============
(3) warning: constant in conditional context
(4) warning: constant in conditional context
(5) warning: constant in conditional context
```

`constant operand to op: !`

Format: Simple

The operand of the `NOT` operator was a constant. Preceding the statement with `/* CONSTCOND */` suppresses the message for that statement; invoking lint with `-h` suppresses it for every statement.

```
1 void fun()
2 {
3   if (! 0) return;
4   /* CONSTCOND */
5   if (! 0) return;
6 }
============
(3) warning: constant operand to op: "!"
```

`constant truncated by assignment`

Format: Simple

An integral constant expression was assigned or returned to an object of an integral type that cannot hold the value without truncation.

```
1 unsigned char f()
2 {
3   unsigned char i;
4   i = 255;
5   i = 256;
6   return 256;
7 }
============
(5) warning: constant truncated by assignment
(6) warning: constant truncated by assignment
```

conversion of pointer loses bits
Format: Simple
A pointer was assigned to an object of an integral type that is smaller than the pointer.

```
1 void fun()
2 {
3   int j = 100;
4   int *i;
5   i = &j;
6   c = i;
7 }
============
(6) warning: conversion of pointer loses bits
```

conversion to larger integral type may sign-extend incorrectly
Format: Compound
A variable of type "plain" `char` was assigned to a variable of a larger integral type. Whether a "plain" `char` is treated as signed or unsigned is implementation-defined. The message is issued only when lint is invoked with `-p`, and is suppressed when it is invoked with `-a`.

```
1 void fun()
2 {
3   char c = 0;
4   short s = 0;
5   long l;
6   l = c;
7   l = s;
8 }
============
conversion to larger integral type may sign-extend incorrectly
    (6)
```

declaration is unused in block
Format: Compound
An external variable or function was declared but not used in an inner block.

```
1 int fun()
2 {
3   int foo();
4   int bar();
5   return foo();
6 }
============
declaration unused in block
    (4) bar
```

`declared global, could be static`
Format: Compound
An external variable or function was declared global, that is, not declared `static`, but was referenced only in the file in which it was defined. The message is suppressed when lint is invoked with `-m`.

```
1   int i;
2   static int bar()
3   {
4       return i;
5   }
6   int foo()
7   {
8       return i;
9   }
10  main()
11  {
12      int a;
13      a = foo();
14      a = bar();
15      return a;
16  }
=============
declared global, could be static
    i       file.c(1)
    foo     file.c(7)
```

`equality operator == found where = was expected`
Format: Simple
An equality operator was found where a side effect was expected.

`1 void fun(a, b)`
`2 int a, b;`
`3 {`
`4   a == b;`
`5   for (a == b; a < 10; a++);`
`6 }`
`============`
`(4) warning: equality operator "==" found where "=" was expected`
`(5) warning: equality operator "==" found where "=" was expected`

`evaluation order undefined:` *name*
Format: Simple
A variable was changed by a side effect and used elsewhere in the same expression.

`1 int a[10];`
`2 main()`
`3 {`
`4   int i = 1;`
`5   a[i++] = i;`
`6 }`
`============`
`(5) warning: evaluation order undefined: i`

fallthrough on case statement
Format: Simple
Execution fell through one `case` to another without a `break` or `return`. Preceding a `case` statement with `/* FALLTHRU */`, or `/* NOTREACHED */` when the `case` cannot be reached from the preceding `case` (see below), suppresses the message for that statement; invoking lint with `-h` suppresses it for every statement.

```
1 void fun(int i)
2 {
3   switch (i) {
4   case 10:
5       i = 0;
6   case 12:
7       return;
8   case 14:
9       break;
10  case 15:
11  case 16:
12      break;
13  case 18:
14      i = 0;
15      /* FALLTHRU */
16  case 20:
17      error("bad number");
18      /* NOTREACHED */
19  case 22:
20      return;
21  }
22 }
============
(6) warning: fallthrough on case statement
```

`function argument ( number ) declared inconsistently`

Format: Compound

The parameter types in a function prototype declaration or definition differed from their types in another declaration or definition. The message described after this one is issued for uses (not declarations or definitions) of a prototype with the wrong parameter types.

```
file i3a.c
1 int fun1(int);
2 int fun2(int);
3 int fun3(int);
file i3b.c
1 int fun1(int *i);
2 int fun2(int *i) {}
3 void foo()
4 {
5   int *i;
6   fun3(i);
7 }
============
function argument ( number ) declared inconsistently
    fun2 (arg 1)        i3b.c(2) int * :: i3a.c(2) int
    fun1 (arg 1)        i3a.c(1) int :: i3b.c(1) int *
function argument ( number ) used inconsistently
    fun3 (arg 1)        i3a.c(3) int :: i3b.c(6) int *
```

function argument *number* used inconsistently
Format: Compound
The argument types in a function call did not match the types of the formal parameters in the function definition. (And see the discussion of the preceding message.)

```
file f1.c
1 int fun(int x, int y)
2 {
3   return x + y;
4 }
file f2.c
1 int main()
2 {
3   int *x;
4   extern int fun();
5   return fun(1, x);
6 }
============
function argument 2 used inconsistently
    fun( arg 2 )     f1.c(2) int :: f2.c(5) int *
```

function argument type inconsistent with format
Format: Compound
An argument was inconsistent with the corresponding conversion specification in the control string of a `[fs]printf()` or `[fs]scanf()` function call. (See also `/* PRINTFLIKE`*n*` */` and `/* SCANFLIKE`*n*` */` in the list of directives in "Usage" on page 265.)

```
1 #include <stdio.h>
2 main()
3 {
4 int i;
5 printf("%s", i);
6 }
============
function argument type inconsistent with format
    printf(arg 2) int :: (format) char * test.c(5)
```

`function called with variable number of arguments`

Format: Compound

A function was called with the wrong number of arguments. Preceding a function definition with `/* VARARGS`*n*` */` suppresses the message for calls with *n* or more arguments; defining and declaring a function with the ANSI C notation "..." suppresses it for every argument. (And see the discussion of the message following this one.)

```
file f1.c
1 int fun(int x, int y, int z)
2 {
3   return x + y + z;
4 }
5 int fun2(int x, ...)
6 {
7   return x;
8 }
10 /* VARARGS1 */
11 int fun3(int x, int y, int z)
12 {
13  return x;
14 }
file f2.c
1 int main()
2 {
3   extern int fun(), fun3(), fun2(int x, ...);
4   return fun(1, 2);
5   return fun2(1, 2, 3, 4);
6   return fun3(1, 2, 3, 4, 5);
7 }
============
function called with variable number of arguments
    fun f1.c(2) :: f2.c(4)
```

`function declared with variable number of arguments`
Format: Compound
The number of parameters in a function prototype declaration or definition differed from their number in another declaration or definition. Declaring and defining the prototype with the ANSI C notation "..." suppresses the warning if all declarations have the same number of arguments. The message immediately preceding this one is issued for uses (not declarations or definitions) of a prototype with the wrong number of arguments.

```
file i3a.c
1 int fun1(int);
2 int fun2(int);
3 int fun3(int);
file i3b.c
1 int fun1(int, int);
2 int fun2(int a, int b) {}
3 void foo()
4 {
5   int i, j, k;
6   i = fun3(j, k);
7 }
============
function declared with variable number of arguments
    fun2      i3a.c(2) :: i3b.c(2)
    fun1      i3a.c(1) :: i3b.c(1)
function called with variable number of arguments
    fun3      i3a.c(3) :: i3b.c(6)
```

`function falls off bottom without returning value`
Format: Compound
A nonvoid function did not return a value to the invoking function. If the closing curly brace is truly not reached, preceding it with `/* NOTREACHED */` suppresses the message.

`1 fun()`
`2 {}`
`3 void fun2()`
`4 {}`
`5 foo()`
`6 {`
`7  exit(1);`
`8  /* NOTREACHED */`
`9 }`
`============`
`function falls off bottom without returning value`
`  (2) fun`

`function must be of type int:  main()`
Format: Simple
You used a `main()` that did not return `int`, in violation of ANSI C restrictions. The message is issued only when `lint` is invoked with `-Xc`.

`1 void main()`
`2 {}`
`============`
`(2) warning: function must be of type int: main()`

`function returns pointer to [automatic/parameter]`
Format: Compound
A function returned a pointer to an automatic variable or a parameter. Since an object with automatic storage duration is no longer guaranteed to be reserved after the end of the block, the value of the pointer to that object will be indeterminate after the end of the block.

```
1 int *fun(int x)
2 {
3   int a[10];
4   int b;
5   if (x == 1)
6       return a;
7   else if (x == 2)
8       return &b;
9   else return &x;
10 }
============
(6) warning: function returns pointer to automatic
(8) warning: function returns pointer to automatic
(9) warning: function returns pointer to parameter
```

function returns value that is always ignored
Format: Compound
A function contained a `return` statement and every call to the function ignored its return value.

```
file f1.c
1 int fun()
2 {
3   return 1;
4 }
file f2.c
1 extern int fun();
2 int main()
3 {
4   fun();
5   return 1;
6 }
============
function returns value that is always ignored
    fun
```

`function returns value that is sometimes ignored`
Format: Compound
A function contained a `return` statement and some, but not all, calls to the function ignored its return value.

file f1.c

```
1 int fun()
2 {
3   return 1;
4 }
```

file f2.c

```
1 extern int fun();
2 int main()
3 {
4   if(1) {
5       return fun();
6   }
7   else {
8       fun();
9       return 1;
10  }
11 }
============
function returns value that is sometimes ignored
    fun
```

`function valued is used, but none returned`

Format: Compound

A nonvoid function did not contain a `return` statement, yet was used for its value in an expression.

file f1.c

```
1 extern int fun();
2 main()
3 {
4   return fun();
5 }
```

file f2.c

```
1 int fun()
2 {}
============
function value is used, but none returned
    fun
```

`logical expression always false:  op &&`

Format: Simple

A logical `AND` expression checked for equality of the same variable to two different constants, or had the constant 0 as an operand. In the latter case, preceding the expression with `/* CONSTCOND */` suppresses the message.

```
 1 void fun(a)
2 int a;
3 {
4   a = (a == 1) && (a == 2);
5   a = (a == 1) && (a == 1);
6   a = (1 == a) && (a == 2);
7   a = (a == 1) && 0;
8 /* CONSTCOND */
9    a = (0 && (a == 1));
10 }
============
(4) warning: logical expression always false: op "&&"
(6) warning: logical expression always false: op "&&"
(7) warning: logical expression always false: op "&&"
```

`logical expression always true: op ||`

Format: Simple

A logical `OR` expression checked for inequality of the same variable to two different constants, or had a nonzero integral constant as an operand. In the latter case, preceding the expression with `/* CONSTCOND */` suppresses the message.

```
1 void fun(a)
2 int a;
3 {
4   a = (a != 1) || (a != 2);
5   a = (a != 1) || (a != 1);
6   a = (1 != a) || (a != 2);
7   a = (a == 10) || 1;
8 /* CONSTCOND */
9   a = (1 || (a == 10));
10 }
============
(4) warning: logical expression always true: op "||"
(6) warning: logical expression always true: op "||"
(7) warning: logical expression always true: op "||"
```

`malformed format string`

Format: Compound

A `[fs]printf()` or `[fs]scanf()` control string was formed incorrectly. (See also `/* PRINTFLIKE`*n*` */` and `/* SCANFLIKE`*n*` */` in the list of directives in "Usage" on page 265.)

```
1 #include <stdio.h>
2 main()
3 {
4    printf("%y");
5 }
============
malformed format string
    printf test.c(4)
```

may be indistinguishable due to truncation or case
Format: Compound
External names in your program may be indistinguishable when it is ported to another machine because of implementation-defined restrictions as to length or case. The message is issued only when lint is invoked with `-Xc` or `-p`. Under `-Xc`, external names are truncated to the first 6 characters with one case, in accordance with the ANSI C lower bound; under `-p`, to the first 8 characters with one case.

file f1.c
`1 int foobar1;`
`2 int FooBar12;`
file f2.c
`1 int foobar2;`
`2 int FOOBAR12;`
`============`
under –p
`may be indistinguishable due to truncation or case`
`FooBar12 f1.c(2) :: FOOBAR12 f2.c(2)`
under –Xc
`may be indistinguishable due to truncation or case`
`foobar1 f1.c(1) :: FooBar12 f1.c(2)`
`foobar1 f1.c(1) :: foobar2 f2.c(1)`
`foobar1 f1.c(1) :: FOOBAR12 f2.c(2)`

name declared but never used or defined
Format: Compound
A nonstatic external variable or function was declared but not used or defined in any file. The message is suppressed when lint is invoked with `-x`.

file f.c
`1 extern int fun();`
`2 static int foo();`
`============`
`name declared but never used or defined`
`    fun f.c(1)`

name defined but never used
Format: Compound
A variable or function was defined but not used in any file. The message is suppressed when lint is invoked with `-u`.

file f.c
`1 int i, j, k = 1;`
`2 main()`
`3 {`
`4   j = k;`
`5 }`
`============`
`name defined but never used`
`    i f.c(1)`

name multiply defined
Format: Compound
A variable was defined in more than one source file.

file f1.c
`1 char i = 'a';`
file f2.c
`1 long i = 1;`
`============`
`name multiply defined`
`    i f1.c(1) :: f2.c(1)`

name used but not defined
Format: Compound
A nonstatic external variable or function was declared but not defined in any file. The message is suppressed when lint is invoked with `-u`.

file f.c
`1 extern int fun();`
`2 int main()`
`3 {`
`4  return fun();`
`5 }`
`============`
`name used but not defined`
`  fun f.c(4)`

nonportable bit-field type
Format: Compound
You used a bit-field type other than signed `int` or unsigned `int`. The message is issued only when lint is invoked with `-p`. Note that these are the only portable bit-field types. ANSI C supports `int`, `char`, `short`, and `long` bit-field types that may be unsigned, signed, or "plain." It also supports the `enum` bit-field type.

`1 struct u {`
`2 unsigned v:1;`
`3 int w:1;`
`4 char x:8;`
`5 long y:8;`
`6 short z:8;`
`7 };`
`============`
`(3) warning: nonportable bit-field type`
`(4) warning: nonportable bit-field type`
`(5) warning: nonportable bit-field type`
`(6) warning: nonportable bit-field type`

nonportable character constant
Format: Simple
A multi-character character constant in your program may not be portable. The message is issued only when lint is invoked with `-Xc` or `-p`.

`1 int c = 'abc';`
`============`
`(1) warning: nonportable character constant`

only 0 or 2 parameters allowed: main()
Format: Simple
The function `main()` in your program was defined with only one parameter or more than two parameters, in violation of the ANSI C requirement. The message is issued only when lint is invoked with `-Xc`.

`1 main(int argc, char **argv, char **envp)`
`2 {}`
`============`
`(2) warning: only 0 or 2 parameters allowed: main()`

pointer cast may result in improper alignment
Format: Compound
You cast a pointer to one object type to a pointer to an object type with stricter alignment requirements. Doing so may result in a value that is invalid for the second pointer type. The warning is suppressed when `lint` is invoked with `-h`.

`1 void fun()`
`2 {`
`3  short *s;`
`4  int *i;`
`5  i = (int *) s;`
`6 }`
`============`
`pointer cast may result in improper alignment`
`  (5)`

`pointer casts may be troublesome`

Format: Compound

You cast a pointer to one object type to a pointer to a different object type. The message is issued only when lint is invoked with `-p`, and is not issued for the generic pointer `void *`.

```
1 void fun()
2 {
3   int *i;
4   char *c;
5   void *v;
6   i = (int *) c;
7   i = (int *) v;
8 }
============
warning: pointer casts may be troublesome
    (6)
```

`precedence confusion possible:  parenthesize`

Format: Simple

You did not parenthesize an expression that mixes a logical and a bitwise operator. The message is suppressed when lint is invoked with `-h`.

```
1 void fun()
2 {
3   int x = 0, m = 0, MASK = 0, i;
4   i = (x + m == 0);
5   i = (x & MASK == 0); /* eval'd (x & (MASK == 0)) */
6   i = (MASK == 1 & x); /* eval'd ((MASK == 1) & x) */
7 }
============
(5) warning: precedence confusion possible; parenthesize
(6) warning: precedence confusion possible; parenthesize
```

`precision lost in bit-field assignment`

Format: Simple

A constant was assigned to a bit-field too small to hold the value without truncation. Note that in the following example the bit-field `z` may have values that range from 0 to 7 or –4 to 3, depending on the machine.

```
1 void fun()
2 {
3   struct {
4   signed x:3; /* max value allowed is 3 */
5   unsigned y:3; /* max value allowed is 7 */
6   int z:3; /* max value allowed is 7 */
7   } s;
8   s.x = 3;
9   s.x = 4;
10  s.y = 7;
11  s.y = 8;
12  s.z = 7;
13  s.z = 8;
14 }
============
(9) warning: precision lost in bit-field assignment: 4
(11) warning: precision lost in bit-field assignment: 0x8
(13) warning: precision lost in bit-field assignment: 8
```

`set but not used in function`

Format: Compound

An automatic variable or a function parameter was declared and set but not used in a function.

```
1 void fun(y)
2 int y;
3 {
4   int x;
5   x = 1;
6   y = 1;
7 }
============
set but not used in function
    (4) x in fun
    (1) y in fun
```

`statement has no consequent: else`

Format: Simple

An `if` statement had a null else part. Inserting `/* EMPTY */` between the `else` and semicolon suppresses the message for that statement; invoking `lint` with `-h` suppresses it for every statement.

```
1 void f(a)
2 int a;
3 {
4   if (a)
5       return;
6   else;
7 }
============
(6) warning: statement has no consequent: else
```

`statement has no consequent: if`
Format: Simple
An `if` statement had a null `if` part. Inserting `/* EMPTY */` between the controlling expression of the `if` and semicolon suppresses the message for that statement; invoking `lint` with `-h` suppresses it for every statement.

```
1 void f(a)
2 int a;
3 {
4   if (a);
5   if (a == 10)
6 /* EMPTY */;
7   else return;
8 }
============
(4) warning: statement has no consequent: if
```

`statement has null effect`
Format: Compound
An expression did not generate a side effect where a side effect was expected. Note that the message is issued for every subsequent sequence point that is reached at which a side effect is not generated.

```
1 void fun()
2 {
3   int a, b, c, x;
4   a;
5   a == 5;
6   ;
7   while (x++ != 10);
8       (a == b) && (c = a);
9 (     a = b) && (c == a);
10      (a, b);
11 }
============
statement has null effect
    (4) (5) (9) (10)
```

statement not reached
Format: Compound
A function contained a statement that cannot be reached. Preceding an unreached statement with `/* NOTREACHED */` suppresses the message for that statement; invoking lint with `-b` suppresses it for every unreached `break` and empty statement. Note that this message is also issued by the compiler but cannot be suppressed.

```
1 void fun(a)
2 {
3   switch (a) {
4   case 1:
5       return;
6       break;
7   case 2:
8       return;
9 /* NOTREACHED */
10      break;
11  }
12 }
============
statement not reached
    (6)
```

static unused
Format: Compound
A variable or function was defined or declared `static` in a file but not used in that file. Doing so is probably a programming error because the object cannot be used outside the file.

```
1 static int x;
2 static int main() {}
3 static int foo();
4 static int y = 1;
============
static unused
    (4) y (3) foo (2) main (1) x
```

`suspicious comparison of char with` *value*: op *op*
Format: Simple
A comparison was performed on a variable of type "plain" `char` that implied it may have a negative value (< 0, <= 0, >= 0, > 0). Whether a "plain" `char` is treated as signed or nonnegative is implementation-defined. The message is issued only when `lint` is invoked with `-p`.

```
1 void fun(c, d)
2 char c;
3 signed char d;
4 {
5   int i;
6   i = (c == -5);
7   i = (c < 0);
8   i = (d < 0);
9 }
============
(6) warning: suspicious comparison of char with negative
    constant: op "=="
(7) warning: suspicious comparison of char with 0: op "<"
```

`suspicious comparison of unsigned with` *value*: op *op*

Format: Simple

A comparison was performed on a variable of `unsigned` type that implied it may have a negative value (< 0, <= 0, >= 0, > 0).

```
1 void fun(x)
2 unsigned x;
3 {
4   int i;
5   i = (x > -2);
6   i = (x < 0);
7   i = (x <= 0);
8   i = (x >= 0);
9   i = (x > 0);
10  i = (-2 < x);
11  i = (x == -1);
12  i = (x == -1U);
13 }
============
(5) warning: suspicious comparison of unsigned with negative
    constant: op ">"
(6) warning: suspicious comparison of unsigned with 0: op "<"
(7) warning: suspicious comparison of unsigned with 0: op "<="
(8) warning: suspicious comparison of unsigned with 0: op ">="
(9) warning: suspicious comparison of unsigned with 0: op ">"
(10) warning: suspicious comparison of unsigned with negative
    constant: op "<"
(11) warning: suspicious comparison of unsigned with negative
    constant: op "=="
```

too few arguments for format
Format: Simple
A control string of a `[fs]printf()` or `[fs]scanf()` function call had more conversion specifications than there were arguments remaining in the call. (See also `/* PRINTFLIKE`*n*` */` and `/* SCANFLIKE`*n*` */` in the list of directives in "Usage" on page 265.)

```
1 #include <stdio.h>
2 main()
3 {
4   int i;
5   printf("%d%d", i);
6 }
============
too few arguments for format
    printf test.c(5)
```

too many arguments for format
Format: Compound
A control string of a `[fs]printf()` or `[fs]scanf()` function call had fewer conversion specifications than there were arguments remaining in the call. (See also `/* PRINTFLIKE`*n*` */` and `/* SCANFLIKE`*n*` */` in the list of directives in "Usage" on page 265.)

```
1 #include <stdio.h>
2 main()
3 {
4   int i, j;
5   printf("%d", i, j);
6 }
============
too many arguments for format
    printf test.c(5)
```

`value type declared inconsistently`
Format: Compound
The return type in a function declaration or definition did not match the return type in another declaration or definition of the function. The message is also issued for inconsistent declarations of variable types.

file f1.c
`1 void fun() {}`
`2 void foo();`
`3 extern int a;`
file f2.c
`1 extern int fun();`
`2 extern int foo();`
`3 extern char a;`
`============`
`value type declared inconsistently`
`fun f1.c(1) void() :: f2.c(1) int()`
`foo f1.c(2) void() :: f2.c(2) int()`
`a f1.c(3) int :: f2.c(3) char`

`value type used inconsistently`

Format: Compound

The return type in a function call did not match the return type in the function definition.

file f1.c

```
1 int *fun(p)
2 int *p;
3 {
4   return p;
5 }
```

file f2.c

```
1 main()
2 {
3   int i, *p;
4   i = fun(p);
5 }
============
value type used inconsistently
    fun f1.c(3) int *() :: f2.c(4) int()
```

`variable may be used before set:` *name*

Format: Simple

The first reference to an automatic, non-array variable occurred at a line number earlier than the first assignment to the variable. Note that taking the address of a variable implies both a set and a use, and that the first assignment to any member of a struct or union implies an assignment to the entire struct or union.

```
1 void fun()
2 {
3   int i, j, k;
4   static int x;
5   k = j;
6   i = i + 1;
7   x = x + 1;
8 }
============
(5) warning: variable may be used before set: j
(6) warning: variable may be used before set: i
```

varable unused in function
Format: Compound
A variable was declared but never used in a function.

1 void fun()
2 {
3 int x, y;
4 static z;
5 }
==========
variable unused in function
  (4) z in fun
  (3) y in fun
  (3) x in fun

Part 3—Appendices

ANSI C Data Representations

This appendix describes how ANSI C represents data in storage and the mechanisms for passing arguments to functions. This chapter is intended as a guide to programmers who wish to write or use modules in languages other than C and have those modules interface to C code.

A.1 Storage Allocation

Table A-1 Storage Allocation for Data Types

Data Type	Internal Representation
`char` elements	a single 8-bit byte aligned on a byte boundary.
`short` integers	half word (two bytes or 16 bits), aligned on a two-byte boundary.
`int` and `long`	32 bits (four bytes or one word), aligned on a four-byte boundary.
`long long`[a]	64 bits (8 bytes, 2 words), aligned on a double-word (eight-byte) boundary.
`float`	32 bits (four bytes or one word), aligned on a four-byte boundary. A `float` has a sign bit, 8-bit exponent, and 23-bit fraction.
`double`	64 bits (eight bytes or two words), aligned on a double-word boundary. A `double` element has a sign bit, an 11-bit exponent and a 52-bit fraction.

a. `long long` is not available in `-Xc` mode.

A.2 Data Representations

Bit numberings of any given data element depend on the architecture in use: Sun-4s and SPARCStations use bit 0 as the least significant bit, with byte 0 being the most significant byte. The tables below describe the various representations.

Integer Representations

Integer types used in ANSI C are `short`, `int`, `long`, and `long long`:[1]

Table A-2 Representation of `short`

Bits	Content
`8-15`	Byte 0
`0-7`	Byte 1

Table A-3 Representation of `int` and `long`

Bits	Content
24-31	Byte 0
16-23	Byte 1
8-15	Byte 2
0-7	Byte 3

Table A-4 Representation of `long long`

Bits	Content
56 - 63	Byte 0
48 - 55	Byte 1
40- 48	Byte 2
32 - 39	Byte 3
24 - 31	Byte 4
16 - 23	Byte 5
8 - 15	Byte 6
0 - 7	Byte 7

`float` *and* `double` *Representation*

`float` and `double` data elements are represented according to the "ANSI IEEE" 754-1985 standard. The tables below,

s
= sign (1 bit)

e
= biased exponent (11bits)

1. `long long` is not available in `-Xc` mode.

f
= fraction (23 bits)

u
= unsigned

Table A-5 `float` Representation

Bits	Name	Content
`31`	Sign	1 iff number is negative.
`23-30`	Exponent	Eight-bit exponent, biased by 127. Values of all zeros, and all ones, reserved.
`0-22`	Fraction	23-bit fraction component of normalized significand. The "one" bit is "hidden".

Table A-6 `double` Representation

Bits	Name	Content
`63`	Sign	1 iff number is negative.
`52-62`	Exponent	Eleven-bit exponent, biased by 1023. Values of all zeros, and all ones, reserved.
`0-51`	Fraction	52-bit fraction component of normalized significand. The "one" bit is "hidden".

A `float` or `double` number is represented by the form:

$$(-1)^{Sign} 2^{(exponent-bias)} 1.f$$

where "1.*f*" is the significand and "*f*" is the bits in the significand fraction.

Extreme Number Representation

Normalized `float` and `double` numbers are said to contain a "hidden" bit, providing for one more bit of precision than would otherwise be the case.

Table A-7 `float` Representations

`normalized number (0<e<255):`	$(-1)^{Sign} 2^{(exponent - 127)} 1.f$
`subnormal number (e=0, f!=0):`	$(-1)^{Sign} 2^{(126)} 1.f$
`zero (e=0, f=0):`	$(-1)^{Sign} 0$

`signaling NaN`
 	
`Quiet NaN`
`Infinity`
	s=u, e=255(max); f=.0uuu-uu (at least one bit must be non-zero)
	s=u, e=255(max); f=.1uuu-uu
	s=u, e=255(max); f=.0000-00 (all zeroes)

Table A-8 `double` Representations

`normalized number (0<e<2047):`	$(-1)^{Sign} 2^{(exponent - 1023)} 1.f$
`subnormal number (e=0, f!=0):`	$(-1)^{Sign} 2^{(1022)} 1.f$
`zero (e=0, f=0):`	$(-1)^{Sign} 0$
`signaling NaN`	s=u, e=2047(max); f=.0uuu-uu (at least one bit must be non-zero)
`Quiet NaN`	s=u, e=2047(max); f=.1uuu-uu
`Infinity`	s=u, e=2047(max); f=.0000-00 (all zeroes)

Hexadecimal Representation of Selected Numbers

Table A-9 Hexadecimal Representation of Selected Numbers

Value	float	double
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxx

Pointer Representation

A pointer in C occupies four bytes. The `NULL` value pointer is equal to zero.

Array Storage

Arrays are stored with their elements in a specific storage order. The elements are actually stored in a linear sequence of storage elements.

C arrays are stored in row-major order; the last subscript in a multi-dimensional array varies fastest.

String data types are simply arrays of `char` elements.

Arithmetic Operations on Extreme Values

This subsection describes the results derived from applying the basic arithmetic operations to combinations of extreme and ordinary floating-point values.

No traps or any other exception actions are taken.

All inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen. In all the tables below, the abbreviations have the following meanings:

Table A-10 Extreme Values Usage

Abbreviation	Meaning
`Num`	Subnormal or Normalized Number
`Inf`	Infinity (positive or negative)
`NaN`	Not a Number
`Uno`	Unordered

The tables that follow describe the types of values that result from arithmetic operations performed with combinations of different types of operands.

Table A-11 Addition and Subtraction Results

Addition and Subtraction				
Left Operand	**Right Operand**			
	0	**Num**	**Inf**	**NaN**
0	0	Num	Inf	NaN
Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Note	NaN
NaN	NaN	NaN	NaN	NaN

Note – Inf + Inf = Inf; Inf - Inf = NaN

Table A-12 Multiplication Results

Multiplication				
Left Operand	**Right Operand**			
	0	**Num**	**Inf**	**NaN**
0	0	0	NaN	NaN
Num	0	Num	Inf	NaN
Inf	NaN	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN

Table A-13 Division Results

Division				
Left Operand	**Right Operand**			
	0	**Num**	**Inf**	**NaN**
0	NaN	0	0	NaN
Num	Inf	Num	0	NaN
Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN

Table A-14 Comparison Results

	Comparison			
Left Operand	**Right Operand**			
	0	**Num**	**Inf**	**NaN**
0	=	<	<	Uno
Num	>		<	Uno
Inf	>	>		Uno
NaN	Uno	Uno	Uno	Uno

Note – NaN compared with NaN is Unordered, and also results in inequality. +0 compares equal to -0.

A.3 *Argument Passing Mechanism*

This section describes how arguments are passed in ANSI C.

All arguments to C functions are passed by value.

Actual arguments are pushed onto the stack in the reverse order from which they are declared in a function declaration.

Actual arguments which are expressions are evaluated before the function reference. The result of the expression is then pushed onto the stack.

Functions return `integer` and `float` results in register `%o0` , while `double` results are returned in `%f0` and `%f1` .

`long long`[1] integers are *passed* in registers with the higher word order in `%oN` and the lower order word in `%o(N+1)`. In-register results are *returned* in `%i0` and `%i1`, with similar ordering.

1. Not available in `-Xc` mode.

All arguments, except `doubles`, are passed as four-byte values; a `double` is passed as an eight-byte value.[1]

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack.

A.4 Referencing Data Objects in C

This section describes how variables of different types are actually accessed (or referenced). The method and notations of access, of course, differ depending on whether the object is a simple variable, an array, a structure, or a union.

Referencing Simple Variables

A plain variable (of simple scalar type) is accessed by its identifier. Since such a simple variable has no structure, its identifier alone is enough to reference it.

```
#include <stdio.h>

int egress;
float lightly;
char coal;
extern double sin();

main()
{
    egress = 10;
    coal = 'a';
    lightly = 3.14;

    putc(coal,stdout);
    (void) printf(" %f\n",sin(lightly));

    return 0;
}
```

Figure A-1 Examples of Simple Variable References

1. In previous versions of C, all `float` values were passed as `doubles`.

Referencing With Pointers

A variable can also be declared as a pointer to another object. In this case, the reference to the object must be done with the pointer notation. Placing an asterisk character * in front of an identifier uses that identifier as a pointer to an object, and the thing that is read from or written to is the object that the identifier points to.

```
#include <stdio.h>

int *egress;
float *lightly;
char *coal;
extern double sin();

main()
{

    egress = (int *) malloc(sizeof(egress));
    *egress = 10;

    lightly = (float *) malloc(sizeof(lightly));
    *lightly = 3.14;

    coal = "Hello Mateo";

    (void) printf("%f\n",sin(*lightly));
    (void) printf("%s\n",coal);

    return 0;
}
```

Figure A-2 Examples of Pointer References

Referencing Array Elements

When an identifier of an array type appears in an expression, the identifier is converted to a pointer to the first member of the array.

The subscript operation [] is interpreted such that

```
E1 [E2]
```

is equivalent to the construct

```
*((E1) + (E2))
```

```
#include <stdio.h>

int egress[10];
float lightly[5][5];
char coal[15];
extern double sin();
int idx;
int idy;

main()
{

    for (idx = 0; idx < 10; idx++)
       egress[idx] = idx;

    for (idx = 0; idx < 5; idx++)
       for (idy = 0; idy < 5; idy++)
          (void) printf("%f\n", sin(lightly[idx][idy]));

    for (idx = 0; idx < 12; idx++)
       (void) printf("%c",coal[idx]);
    printf("\n");

    return 0;
}
```

Figure A-3 Examples of Array Variable References

Referencing Structures and Unions

There are only three operations which may be done on a structure or a union:

1. A member of the structure or union can be referenced by means of the . or -> operator.
2. The address of the entire structure or union can be taken, with the & operator.

3. One structure can be copied to another of the same type with the assignment operator.

The . operator is used in contexts where the structure or union identifier is available directly to the expression. The -> operator is used when the identifier for the structure or union is a pointer to the object. Structures can also be passed as parameters, returned from functions, or assigned to variables of the same structure or union type.

```
#define MAXLEN 256
#define NULL 0

struct vallist {
    char *name;
    char valtype;
    int value;
    struct vallist *nextval;
};

struct vallist  *
demo(char *wanted, struct vallist *valhead)
{
    int i;
    struct {
        int level;
        char *cp;
        char pbuffer[MAXLEN];
    } putter;

    struct vallist *pointer;

    putter.level = 10;
    for (i = 0; i < MAXLEN; i++)
        putter.pbuffer[i] = *putter.cp;

    for (pointer = valhead; pointer != NULL; pointer = pointer-
>nextval)
        if (strcmp(pointer->name, wanted) == 0)
            return(pointer);

}
```

Figure A-4 Examples of Accessing Members of Structures

Implementation-Defined Behavior

The *American National Standard for Information Systems — Programming Language C*, X3.159-1989 defines the behavior of ANSI-conformant C. However, this standard leaves a number of issues as "implementation-defined," that is, as varying from compiler to compiler.

This chapter details these areas. They made be readily compared to the ANSI standard itself:

- Each issue uses the same section text (in ***boldface italic***) as found in the ANSI standard.
- Each issue is preceded by its corresponding section number in the ANSI standard.

Translation

(2.1.1.3) Identification of diagnostics:

Error messages have the following format:

filename, `line` *line number*: *message*

Warning messages have the following format:

filename, `line` *line number*: *warning message*

Where:

- *filename* is the name of the file containing the error or warning

- *line number* is the number of the line on which the error or warning was found
- *message* is the diagnostic message

Environment

(2.1.2.2.1) Semantics of arguments to `main`:

```
int main (int argc, char *argv[])
{
....
}
```

`argc` is the number of command line arguments that the program was invoked with. After any shell expansion, `argc` is always equal to at least 1 (the name of the program).

`argv` is an array of pointers to the command line arguments.

(2.1.2.3) What constitutes an interactive device:

An interactive device is one for which the system library call `isatty()` returns a non-zero value.

Identifiers

(3.1.2) The number of significant initial characters (beyond 31) in an identifier without external linkage:

The first 1023 characters are significant. Identifiers are case-sensitive.

(3.1.2) The number of significant initial characters (beyond 6) in an identifier with external linkage:

The first 1023 characters are significant. Identifiers are case-sensitive.

Characters

(2.2.1) The members of the source and execution character sets, except as explicitly specified in the Standard:

Both sets are identical to the ASCII character sets.

(2.2.1.2) The shift states used for the encoding of multibyte characters:

There are no shift states; `MB_CUR_MAX` is equal to 1.

(2.2.4.2.1) The number of bits in a character in the execution character set:

There are 8 bits in a character.

(3.1.3.4) The mapping of members of the source character set (in character and string literals) to members of the execution character set:

Mapping is identical between source and execution characters.

(3.1.3.4) The value of an integer character constant that contains a character or escape sequence not represented in the basic execution character set or the extended character set for a wide character constant:

It is the numerical value of the rightmost character. For example, `'\q'` will equal `'q'`. A warning will be emitted if such an escape sequence occurs.

(3.1.3.4) The value of an integer character constant that contains more than one character or a wide character constant that contains more than one multibyte character:

It is the numerical value of the rightmost character. For example, `'qq'` will equal `'q'`. A warning will be emitted if such an escape sequence occurs.

(3.1.3.4) The current locale used to convert multibyte characters into corresponding wide characters (codes) for a wide character constant:

The current locale is "C" locale.

(3.2.1.1) Does a plain `char` *have the same range of values as* `signed char` *or* `unsigned char`*:*

A `char` is treated as a `signed char`.

Integers

(3.1.2.5) The representations and sets of values of the various types of integers:

Table B-1 Representations and sets of values of integers

	Bits	Min	Max
`char`	8	-128	127
`signed char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32768	32767
`signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`integer`	32	-2147483648	2147483647
`signed integer`	32	-2147483648	2147483647
`unsigned integer`	32	0	4294967295
`long`	32	-2147483648	2147483647
`signed long`	32	-2147483648	2147483647
`unsigned long`	32	0	4294967295
`long long`[a]	64	-9223372036854775808	9223372036854775807
`signed long long`[a]	64	-9223372036854775808	9223372036854775807
`unsigned long long`[a]	64	0	18446744073709551615

a. Not valid in `-Xc` mode.

(3.2.1.2) The result of converting an integer to a shorter signed integer, or the result of converting an unsigned integer to a signed integer of equal length, if the value cannot be represented:

When an integer is converted to a shorter signed integer, the low order bits will be copied from the longer integer to the shorter signed integer. The result may be negative.

When an unsigned integer is converted to a signed integer of equal size, the low order bits will be copied from the unsigned integer to the signed integer. The result may be negative.

(3.3) The results of bitwise operations on signed integers:

The result of a bitwise operation applied to a signed type is the bitwise operation of the operands, including the sign bit. Thus each bit in the result is set if and only if each of the corresponding bits in both of the operands is set.

(3.3.5) The sign of the remainder on integer division:

The result will be the same sign as the dividend; thus the remainder of -23/4 is -5.

(3.3.7) The result of a right shift of a negative-valued signed integral type:

The result of a right shift will be a signed right shift.

Floating Point

(3.1.2.5) The representations and sets of values of the various types of floating point numbers:

Table B-2 Values of floating-point numbers (Sheet 1 of 2)

	`float`
Bits	32
Min	1.17549435E-38
Max	3.40282347E+38

Table B-2 Values of floating-point numbers (Sheet 2 of 2)

Epsilon	1.19209290E-07
`double`	
Bits	64
Min	2.2250738585072014E-308
Max	1.7976931348623157E+308
Epsilon	2.2204460492503131E-16
`long double`	
Bits	128
Min	3.362103143112093506262677817321752603E-4932
Max	1.189731495357231765085759326628007016E+4932
Epsilon	1.925929944387235853055977942584927319E-34

(3.2.1.3) The direction of truncation when an integral number is converted to a floating-point number that cannot exactly represent the original value:

Numbers are rounded to the nearest value that can be represented.

(3.2.1.4) The direction of truncation or rounding when a floating- point number is converted to a narrower floating-point number:

Numbers are rounded to the nearest value that can be represented.

Arrays And Pointers

(3.3.3.4, 4.1.1) The type of integer required to hold the maximum size of an array; that is, the type of the `sizeof` *'operator,* `size_t`*:*

`unsigned int` as defined in `stddef.h`.

(3.3.4) The result of casting a pointer to an integer or vice versa:

The bit pattern does not change for pointers and values of type `int`, `long`, `unsigned int` and `unsigned long`.

(3.3.6, 4.1.1) The type of integer required to hold the difference between two pointers to members of the same array, `ptrdiff_t`***:***

`int` as defined in `stddef.h`.

Registers

(3.5.1) The extent to which objects can actually be placed in registers by use of the `register` ***storage-class specifier:***

The number of effective register declarations depends on patterns of use and definition within each function and is bounded by the number of registers available for allocation. Neither the compiler nor the optimizer is required to honor register declarations.

Structures, Unions, Enumerations And Bit-Fields

(3.3.2.3) A member of a union object is accessed using a member of a different type:

The bit pattern stored in the union member is accessed, and the value interpreted, according to the type of the member by which it was accessed.

(3.5.2.1) The padding and alignment of members of structures.

Table B-3 Padding and alignment of structure members

type	alignment boundary	byte alignment
`char`	byte	1
`short`	halfword	2
`int`	word	4
`long`	word	4
`float`	word	4
`double`	doubleword	8
`long double`	doubleword	8
`pointer`	word	4
`long long`[a]	doubleword	8

a. Not available in `-Xc` mode.

Structure members are padded internally so that every element is aligned on the appropriate boundary.

Alignment of structures is the same as its more strictly aligned member. For example, a `struct` with only `char`'s would have no alignment restrictions, whereas a `struct` containing a `double` would be aligned on an 8-byte boundary.

(3.5.2.1) Whether a plain `int` *bit-field is treated as a* `signed int` *bit-field or as an* `unsigned int` *bit-field:*

It is treated as an `unsigned int`.

(3.5.2.1) The order of allocation of bit-fields within an `int`*:*

Bit-fields are allocated within a storage unit from high-order to low-order.

(3.5.2.1) Whether a bit-field can straddle a storage-unit boundary:

Bit-fields do not straddle storage-unit boundaries.

(3.5.2.2) The integer type chosen to represent the values of an enumeration type:

This is an int.

Qualifiers

(3.5.3) What constitutes an access to an object that has volatile-qualified type:

Each reference to the name of an object will constitute one access to the object.

Declarators

(3.5.4) The maximum number of declarators that may modify an arithmetic, structure, or union type:

No limit imposed by the compiler.

Statements

(3.6.4.2) The maximum number of `case` ***values in a*** `switch` ***statement:***

No limit imposed by the compiler.

Preprocessing Directives

(3.8.1) Whether the value of a single-character character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set:

A character constant within a preprocessing directive has the same numeric value as it has within any other expression.

(3.8.1) Whether such a character constant may have a negative value:

Character constants in this context may have negative values.

(3.8.2) The method for locating includable source files:

A file whose name is delimited by < > is searched for first in the directories named by the `-I` option and then in the standard directory. The standard directory is `/usr/include`, unless the `-YI` option is used to specify a different default location.

A file whose name is delimited by quotes is searched for first in the directory of the source file that contains the `#include`, then in directories named by the `-I` option, and last in the standard directory.

If a file name enclosed in < > or double quotes begins with a `'/'` character, the file name shall be interpreted as a path name beginning in the root directory. The search for this file will begin in the root directory only.

(3.8.2) The support of quoted names for includable source files:

Quoted filenames in include directives are supported.

(3.8.2) The mapping of source file character sequences:

Source file characters are mapped to their corresponding ASCII values.

(3.8.6) The behavior on each recognized `#pragma` *directive:*

The following pragmas are supported:

`fini` *identifier*
: marking *identifier* as a "finalization function." Such functions are expected to be of type `void` and to accept no arguments, and are called either when a program terminates under program control or when the containing shared object is removed from memory. As with "initialization functions," finalization functions are executed in the order processed by the link editor(s).

`init` *identifier*
: marking *identifier* as an "initialization function." Such functions are expected to be of type `void` and to accept no arguments, and are called while constructing the memory image of the program at the start of

execution. In the case of initializers in a shared object, they will be executed during the operation that brings the shared object into memory, either program start-up or some dynamic loading operation such as `dlopen()`. The only ordering of calls to initialization functions is the order in which they were processed by the link editor(s), both static and dynamic.

`ident` *string*
: place *string* in the `.comment` section of the executable

`int_to_unsigned` *function name*
: For a function that returns a type of `unsigned`, in `-Xt` or `-Xs` mode, change the function return to be of type `int`.

`unknown_control_flow (`*name*`, [,`*name*`] )`
: Specifies a list of routines that violate the usual control flow properties of procedure calls. For example, the statement following a call to `setjmp()` can be reached from an arbitrary call to any other routine. The statement is reached by a call to `longjmp()`. Since such routines render standard flowgraph analysis invalid, routines that call them cannot be safely optimized; hence, they are compiled with the optimizer disabled.

`unshared (`*name* `[,` *name* `])`
: Any identifer named in the id list must be marked in the symbol table as unshared (thread-local), so that subsequent symbol table accesses for the symbol will be able to pass along this information to any tool that needs it. `errno` is an example of a symbol which should be marked.

`weak` *function name* = *_function name*
: If a defined global symbol *function name* exists, the appearance of a weak symbol *_function name* with the same name will not cause an error.

`weak` *function name*
: The linker will not complain if it does not find a definition for *function name*.

The compiler ignores unrecognized pragmas.

(3.8.8) The definitions for `__DATE__` *and* `__TIME__` *when, respectively, the date and time of translation are not available:*

These macros are always available from the environment.

(4.1.5) The null pointer constant to which the macro `NULL` *expands:*

NULL equals 0.

(4.2) The diagnostic printed by and the termination behavior of the `assert` *function:*

The diagnostic is:

`Assertion failed:` *statement*`.  file` *filename*`, line` *number*

Where:

- *statement* is the statement which failed the assertion
- *filename* is the name of the file containing the failure
- *line number* is the number of the line on which the failure occured

(4.3.1) The sets of characters tested for by the `isalnum`, `isalpha`, `iscntrl`, `islower`, `isprint`, *and* `isupper` *functions:*

Table B-4 Character sets tested by `isalpha`, `islower`, etc.

`isalnum`	ASCII characters 'A'-'Z', 'a'-'z' and '0'-'9'
`isalpha`	ASCII characters 'A'-'Z' and 'a'-'z'
`iscntrl`	ASCII characters with value 0-31 and 127.
`islower`	ASCII characters 'a'-'z'
`isprint`	ASCII characters with decimal value 32 through 126
`isupper`	ASCII characters 'A'-'Z'

(4.5.1) The values returned by the mathematics functions on domain errors:

Table B-5 Values returned on domain errors

		Compiler Modes	
error	**math functions**	**-Xs -Xt**	**-Xa -Xc**
DOMAIN	acos(|x|>1)	0.0	0.0
DOMAIN	asin(|x|>1)	0.0	0.0
DOMAIN	atan2(+-0,+-0)	0.0	0.0
DOMAIN	y0(0)	-HUGE	-HUGE_VAL
DOMAIN	y0(x<0)	-HUGE	-HUGE_VAL
DOMAIN	y1(0)	-HUGE	-HUGE_VAL
DOMAIN	y1(x<0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,x<0)	-HUGE	-HUGE_VAL
DOMAIN	log(x<0)	-HUGE	-HUGE_VAL
DOMAIN	log10(x<0)	-HUGE	-HUGE_VAL
DOMAIN	pow(0,0)	0.0	1.0
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL
DOMAIN	pow(neg,non-integal)	0.0	NaN
DOMAIN	sqrt(x<0)	0.0	NaN
DOMAIN	fmod(x,0)	x	NaN
DOMAIN	remainder(x,0)	NaN	NaN
DOMAIN	acosh(x<1)	NaN	NaN
DOMAIN	atanh(|x|>1)	NaN	NaN

(4.5.1) Whether the mathematics functions set the integer expression `errno` *to the value of the macro* `ERANGE` *on underflow range errors:*

Mathematics functions, except `scalbn`, set `errno` to `ERANGE` when underflow is detected.

(4.5.6.4) Whether a domain error occurs or zero is returned when the `fmod` *function has a second argument of zero:*

In this case it returns the first argument with domain error.

Signals

(4.7.1.1) The set of signals for the `signal` *function:*

The semantics for each signal recognized by the `signal` function:

Table B-6 Semantics for `signal` signals

Signal	No.	Default	Event
`SIGHUP`	1	Exit	hangup
`SIGINT`	2	Exit	interrupt
`SIGQUIT`	3	Core	quit
`SIGILL`	4	Core	illegal instruction (not reset when caught)
`SIGTRAP`	5	Core	trace trap (not reset when caught)
`SIGIOT`	6	Core	IOT instruction
`SIGABRT`	6	Core	used by abort
`SIGEMT`	7	Core	EMT instruction
`SIGFPE`	8	Core	floating point exception
`SIGKILL`	9	Exit	kill (cannot be caught or ignored)
`SIGBUS`	10	Core	bus error
`SIGSEGV`	11	Core	segmentation violation
`SIGSYS`	12	Core	bad argument to system call

Table B-6 Semantics for `signal` signals

Signal	No.	Default	Event
SIGPIPE	13	Exit	write on a pipe with no one to read it
SIGALRM	14	Exit	alarm clock
SIGTERM	15	Exit	software termination signal from kill
SIGUSR1	16	Exit	user defined signal 1
SIGUSR2	17	Exit	user defined signal 2
SIGCLD	18	Ignore	child status change
SIGCHLD	18	Ignore	child status change alias
SIGPWR	19	Ignore	power-fail restart
SIGWINCH	20	Ignore	window size change
SIGURG	21	Ignore	urgent socket condition
SIGPOLL	22	Exit	pollable event occured
SIGIO	22	Exit	socket I/O possible
SIGSTOP	23	Stop	stop (cannot be caught or ignored)
SIGTSTP	24	Stop	user stop requested from tty
SIGCONT	25	Ignore	stopped process has been continued
SIGTTIN	26	Stop	background tty read attempted
SIGTTOU	27	Stop	background tty write attempted
SIGVTALRM	28	Exit	virtual timer expired
SIGPROF	29	Exit	profiling timer expired
SIGXCPU	30	Core	exceeded cpu limit
SIGXFSZ	31	Core	exceeded file size limit
SIGWAITINGT	32	Ignore	process's lwps are blocked

(4.7.1.1) The default handling and the handling at program startup for each `signal` recognized by the signal function:

See above.

(4.7.1.1) If the equivalent of `signal(sig, SIG_DFL);` is not executed prior to the call of a signal handler, the blocking of the signal that is performed:

The equivalent of `signal(sig,SIG_DFL)` is always executed.

(4.7.1.1) Whether the default handling is reset if the `SIGILL` signal is received by a handler specified to the 'signal' function:

Default handling is not reset in `SIGILL`.

Streams and Files

(4.9.2) Whether the last line of a text stream requires a terminating new-line character:

The last line does not need to end in a new-line.

(4.9.2) Whether space characters that are written out to a text stream immediately before a new-line character appear when read in:

All characters will appear when the stream is read.

(4.9.2) The number of null characters that may be appended to data written to a binary stream:

No null characters are appended to a binary stream.

(4.9.3) Whether the file position indicator of an append mode stream is initially positioned at the beginning or end of the file:

File position indicator is initially positioned at the end of the file.

(4.9.3) Whether a write on a text stream causes the associated file to be truncated beyond that point:

A write on a text stream will not cause a file to be truncated beyond that point unless a hardware device forces it to happen.

(4.9.3) The characteristics of file buffering:

Output streams, with the exception of the standard error stream (stderr), are by default buffered if the output refers to a file, and line-buffered if the output refers to a terminal. The standard error output stream (stderr) is by default unbuffered.

A buffered output stream saves many characters, and then writes the characters as a block. An unbuffered output stream queues information for immediate writing on the destination file or terminal immediately. Line-buffered output queues each line of output until the line is complete (a newline character is requested).

(4.9.3) Whether a zero-length file actually exists:

A zero-length file does exist in the sense that it has a directory entry.

(4.9.3) The rules for composing valid file names:

A valid file name may be from 1 to 1023 characters in length and may use all character except the characters "null" and slash (/).

(4.9.3) Whether the same file can be open multiple times:

The same file can be opened multiple times.

(4.9.4.1) The effect of the `remove` *function on an open file:*

The file is deleted on the last call which closes the file. A program cannot open a file which has already been removed.

(4.9.4.2) The effect if a file with the new name exists prior to a call to the `rename` *function:*

If the file exists, it is removed and the new file is written over the previously existing file.

(4.9.6.1) The output for `%p` *conversion in the* `fprintf` *function:*

The output for `%p` is equivalent to `%x`.

(4.9.6.2) The input for `%p` *conversion in the* `fscanf` *function:*

The input for `%p` is equivalent to `%x`.

(4.9.6.2) The interpretation of a '-' character that is neither the first nor the last character in the scan list for `%[` *conversion in the* `fscanf` *function:*

The `'-'` character indicates an inclusive range; thus, [0-9] is equivalent to [0123456789].

Errno

(4.9.9.4) The value to which the macro `errno` *is set by the* `fgetpos` *or* `ftell` *function on failure:*

`errno` is set to `EBADF`, `ESPIPE`, or `EINVAL` on failure.

(4.9.10.4) The messages generated by the `perror` *function:*

Table B-7 Error Messages generated by `perror` (Sheet 1 of 5)

Number	Message
1	Not owner
2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Arg list too long
8	Exec format error
9	Bad file number
10	No child processes
11	No more processes
12	Not enough space
13	Permission denied
14	Bad address
15	Block device required
16	Device busy
17	File exists
18	Cross-device link
19	No such device
20	Not a directory
21	Is a directory
22	Invalid argument
23	File table overflow

Table B-7 Error Messages generated by `perror` (Sheet 2 of 5)

Number	Message
24	Too many open files
25	Not a typewriter
26	Text file busy
27	File too large
28	No space left on device
29	Illegal seek
30	Read-only file system
31	Too many links
32	Broken pipe
33	Argument out of domain
34	Result too large
35	No message of desired type
36	Identifier removed
37	Channel number out of range
38	Level 2 not synchronized
39	Level 3 halted
40	Level 3 reset
41	Link number out of range
42	Protocol driver not attached
43	No CSI structure available
44	Level 2 halted
45	Deadlock situation detected/avoided
46	No record locks available
50	Bad exchange descriptor

Table B-7 Error Messages generated by `perror` (Sheet 3 of 5)

Number	Message
51	Bad request descriptor
52	Message tables full
53	Inode table overflow
54	Bad request code
55	Invalid slot
56	File locking deadlock
57	Bad font file format
60	Not a stream device
61	No data available
62	Timer expired
63	Out of stream resources
64	Machine is not on the network
65	Package not installed
66	Object is remote
67	Link has been severed
68	Advertise error
69	Srmount error
70	Communication error on send
71	Protocol error
74	Multihop attempted
77	Not a data message
78	File name too long
79	Value too large for defined data type
80	Name not unique on network

Table B-7 Error Messages generated by `perror` (Sheet 4 of 5)

Number	Message
81	File descriptor in bad state
82	Remote address changed
83	Can not access a needed shared library
84	Accessing a corrupted shared library
85	.lib section in a.out corrupted
86	Attempting to link in more shared libraries than system limit
87	Can not exec a shared library directly
89	Operation not applicable
90	Number of symbolic links encountered during path name traversal exceeds MAXSYMLINKS
93	Directory not empty
94	Too many users
95	Socket operation on non-socket
96	Destination address required
97	Message too long
98	Protocol wrong type for socket
99	Option not supported by protocol
120	Protocol not supported
121	Socket type not supported
122	Operation not supported on transport endpoint
123	Protocol family not supported
124	Address family not supported by protocol family
125	Address already in use
126	Cannot assign requested address

Table B-7 Error Messages generated by `perror` (Sheet 5 of 5)

Number	Message
127	Network is down
128	Network is unreachable
129	Network dropped connection because of reset
130	Software caused connection abort
131	Connection reset by peer
132	No buffer space available
133	Transport endpoint is already connected
134	Transport endpoint is not connected
135	Structure needs cleaning
137	Not a name file
138	Not available
139	Is a name file
140	Remote I/O error
141	Reserved for future use
142	
143	Cannot send after socket shutdown
144	Too many references: cannot splice
145	Connection timed out
146	Connection refused
147	Host is down
148	No route to host
149	Operation already in progress
150	Operation now in progress
151	Stale NFS file handle

Memory

(4.10.3) The behavior of the `calloc`, `malloc`, ***or*** `realloc` ***function if the size requested is zero:***

`Malloc` and `calloc` return a unique pointer if the size is zero. `realloc` will free the object pointed to if the size is zero and the pointer is not null.

`abort` *Function*

(4.10.4.1) The behavior of the `abort` ***function with regard to open and temporary files:***

`abort` first closes all open files, stdio streams, directory streams, and message catalogue descriptors, if possible, and then causes the signal `SIGABRT` to be sent to the calling process.

`exit` *Function*

(4.10.4.3) The status returned by the `exit` ***function if the value of the argument is other than zero,*** `EXIT_SUCCESS`, ***or*** `EXIT_FAILURE`***:***

The value returned by the argument to `exit`.

`getenv` *Function*

(4.10.4.4) The set of environment names and the method for altering the environment list used by the `getenv` ***function:***

The set of environment names provided to a program are the same as those that were in the environment when the program was executed. Any environment variable altered during program execution does not permanently change the environment variable; that is, the environment variable has the same value upon program completion as it did before the program was executed.

`system` *Function*

(4.10.4.5) The contents and mode of execution of the string by the `system` ***function:***

```
(void) execl("/sbin/sh", "sh", (const char *)"-c", string, (char *)0);
```

`strerror` *Function*

(4.11.6.2) The contents of the error message strings returned by the `strerror` ***function:***

See 4.9.10.4 above.

Locale Behavior

(4.12.1) The local time zone and Daylight Savings Time:

The local time zone is set by the environment variable `TZ`.

(4.12.2.1) The era for the `clock` ***function***

The era for the clock is represented as clock ticks with the origin at the beginning of the execution of the program.

Locale-specific Behavior

The following characteristics of a hosted environment are locale-specific:

(2.2.1) The content of the execution character set, in addition to the required members:

There are no extensions to the character set.

(2.2.2) The direction of printing:

Printing is always left to right.

(4.1.1) The decimal-point character:

The decimal point is '.'.

(4.3) The implementation-defined aspects of character testing and case mapping functions:

Same as 4.3.1 above

(4.11.4.4) The collation sequence of the execution character set:

The collation sequence is the ASCII collation sequence.

(4.12.3.5) The formats for time and date:

The names of the months are:

Table B-8 Names of Months

January	May	September
February	June	October
March	July	November
April	August	December

The names of the days of the week are:

Table B-9 Days of the Week

Sunday	Thursday
Monday	Friday
Tuesday	Saturday
Wednesday	

The abbreviated names of the days of the week:

Table B-10 Abbreviated Days of the Week

Sun	Thu
Mon	Fri
Tue	Sat
Wed	

The format for time is:

```
%H:%M:%S
```

The format for date is:

```
%m/%d/%y
```

The format for AM/PM designation is:

```
AM
PM
```

Incompatibilities between ANSI C and Sun C 2.0 (SunOS 4.1.x)

This appendix briefly describes compatibility issues between the ANSI C standard and Sun C 2.0 (available in SunOS 4.1.x).

These compatibility issues include:

- Search paths for header files and libraries.
- `libc` functions.
- `libansi.a` library.
- name space pollution.
- header files modified for SunOS 4.x.
- miscellaneous differences.

C.1 Library Differences

Search Paths

These are the orders in which directories are searched for header files and libraries. These are the default paths; actual paths will depend on your installation. Table C-1 summarizes these paths.

Table C-1 Directory Search Paths

	acc	acc -sys5
header files	/usr/lang/SC2.0/include/cc /usr/include	/usr/lang/SC2.0/include/cc /usr/5include /usr/include
library files	/usr/lang/SC2.0/cg87 /usr/lang/SC2.0 /usr/lib	/usr/lang/SC2.0/cg87 /usr/lang/SC2.0 /usr/5lib /usr/lib

Note – The -sys5 option provides compatibility with /usr/5include header files and /usr/5lib libraries (see Table C-1). To accomplish this compatibility, the driver predefines __SYS5__ when -sys5 is specified.

`libc` *Differences*

Some of the functions in SunOS 4.x `libc` do not behave as specified by the ANSI standard. These differences are detailed in Table C-2.

Table C-2 `libc` Differences (Sheet 1 of 2)

acc	acc -sys5	ANSI standard
`typedef int size_t;`	`typedef int size_t;`	`typedef unsigned int size_t;`
`int abort (void);`	`void abort (void);`	`void abort (void);`
`char *calloc (size_t, size_t);`	`void *calloc (size_t, size_t);`	`void *calloc (size_t, size_t);`
`int clearerr (FILE *);`	`void clearerr (FILE *);`	`void clearerr (FILE *);`
`int fputc (char, FILE *);`	`int fputc (int, FILE *);`	`int fputc (int, FILE *);`
`int free (void *);`	`void free (void *);`	`void free (void *);`
`char *malloc (size_t);`	`void *malloc (size_t);`	`void *malloc (size_t);`

Table C-2 `libc` Differences (Sheet 2 of 2)

acc	acc -sys5	ANSI standard
`char *memchr (char *, int, size_t);`	`char *memchr (char *, int, size_t);`	`void *memchr (const void *, int, size_t);`
`int memcmp (char *, char *, int);`	`int memcmp (char *, char *, int);`	`int memcmp (const void *, const void *, size_t);`
`char *memset (char *, int, int);`	`char *memset (char *, int, int);`	`void *memset (void *, int, size_t);`
`char *memcpy (char *, char *, int);`	`char *memcpy (char *, char *, int);`	`void *memcpy (void *, const void *, size_t);`
`int qsort (void *, size_t, size_t,int (*) (const void *, const void *));`	`void qsort (void *, size_t, size_t,int (*) (const void *, const void *));`	`void qsort (void *, size_t, size_t,int (*) (const void *, const void *));`
`char *realloc (void *, size_t);`	`void *realloc (void *, size_t);`	`void *realloc (void *, size_t);`
`char *sprintf (char *, const char *, ...);`	`int sprintf (char *, const char *, ...);`	`int sprintf (char *, const char *, ...);`
`char *vsprint (char *, const char *, void *);`	`int vsprint (char *, const char *, void *);`	`int vsprint (char *, const char *, void *);`
`int rewind (FILE *);`	`void rewind (FILE *);`	`void rewind (FILE *);`

In addition to the information in Table C-2, `realloc` will not accept a NULL pointer as a first argument. Instead of the NULL pointer being equivalent to a `malloc` call, its use will result in a run-time error.

C.2 *Library* `libansi.a`

`libansi.a` is packaged with C 2.0 (ANSI C) to furnish functionality that is either missing or different on SunOS 4.x, when compared to the ANSI standard. This library is intended to reduce the gap between C 2.0 and ANSI C but not to fulfill all the requirements of the ANSI C standard.

`libansi.a` is located in `/usr/lang/SC2.0`.

Header Files

The following headers in `/usr/lib/SC2.0/include/cc` are packaged with the compiler to support ANSI features:

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

ANSI C Functionality Supplied by libansi.a

The following routines are defined only in `libansi.a`, as they are required by ANSI C specs and are not supplied in SunOS 4.x `libc.a`.

```
atexit ()
difftime ()
div ()
fsetpos ()
labs ()
ldiv ()
memmove ()
raise ()
strerror ()
strtoul ()
```

The following routines are redefined by libansi.a in order to correct non-ANSI behavior of the corresponding routines that are provided by SunOS 4.x or `libc.a`.

```
fflush ()
rand ()
srand ()
```

C.3 Name Space Pollution

The ANSI C Standard reserves certain names for the compiler:

- Identifiers belonging to the list of keywords.
- External names defined in the ANSI Standard library section.
- External names beginning with _.
- All names beginning with _[A-Z}, or __[A-Z, a-z].

All other names are available to the user.

Note – Of the ANSI C Standard's reserved identifiers, internal names beginning with _[a-z] are not considered to be reserved, and are therefore, implicitly available to the programmer.

C 2.0 (ANSI C) does not guarantee all of the above conditions. For example, in `stdio.h`, the internal name `_iob` is referenced by various macros within it (`stdio.h`). In C 2.0 (ANSI C), the programmer should avoid the use of any identifiers beginning with _[A-Z, a-z}, or __[A-Z, a-z].

C.4 Header Files Modified for SunOS 4.x

The following discussion describes problems and their fixes for Sun ANSI C compiler header files modified for SunOS 4.x. The header files are found in /usr/lang/SC2.0/include.

/usr/include/des_crypt.h

Problem:

"/usr/include/des_crypt.h", line 45: warning: old-style declaration; add "int"

"/usr/include/des_crypt.h", line 57: warning: old-style declaration; add "int"

Fix:

Add missing return type for function declaration

/usr/include/hsfs/hsfs_spec.h

Problem:

"/usr/include/hsfs/hsfs_spec.h", line 30: warning: tokens ignored at end of directive line

Fix:

Change #if statement to used defined values

/usr/include/hsfs/hsnode.h

Problem:

"/usr/include/hsfs/hsnode.h", line 84: warning: tokens ignored at end of directive line

Fix:

Change #if statement to used defined values

/usr/include/hsfs/iso_spec.h

Problem:

"/usr/include/hsfs/iso_spec.h", line 25: warning: tokens ignored at end of directive line

Fix:

Change #if statement to used defined values

/usr/include/machine/reg.h
/usr/include/sun4c/reg.h
/usr/include/sun4/reg.h

Problem:

"/usr/include/sun4c/reg.h", line 92: incomplete struct/union/enum fpq: <unnamed>

"/usr/include/sun4c/reg.h", line 92: warning: unnamed union member

"/usr/include/sun4c/reg.h", line 96: (struct) tag redeclared: fpq

Fix:

Move definition of struct fpq before struct fq

/usr/include/mon/eeprom.h

Problem:

Missing semi-colon after last structure member

"/usr/include/mon/eeprom.h", line 249: warning: syntax requires ";" after last struct/union member

Fix:

Add semicolon

/usr/include/rfs/ns_xdr.h

Problem:

"/usr/include/rfs/ns_xdr.h", line 10: warning: comment is replaced by "##"

"/usr/include/rfs/ns_xdr.h", line 11: warning: comment is replaced by "##"

Fix:

Replace /****/ with ##

/usr/include/rfs/rfs_xdr.h

Problem:

"/usr/include/rfs/rfs_xdr.h", line 16: warning: comment is replaced by "##"

"/usr/include/rfs/rfs_xdr.h", line 17: warning: comment is replaced by "##"

Fix:

Replace /****/ with ##

/usr/include/sparc/asm_linkage.h

Problem:

"/usr/include/sparc/asm_linkage.h", line 107: warning: comment is replaced by "##"

Fix:

Replace /****/ with ##

/usr/include/stand/scsi.h

Problem:

"/usr/include/stand/scsi.h", line 216: warning: syntax requires ";" after last struct/union member

"/usr/include/stand/scsi.h", line 216: warning: unnamed struct member

Fix:

Add name followed by semicolon for the structure member

/usr/include/sparc/asm_linkage.h

Problem:

"/usr/include/sun4/asm_linkage.h", line 107: warning: comment is replaced by "##"

Fix:

Replace /****/ with ##

/usr/include/sun4c/asm_linkage.h

Problem:

"/usr/include/sun4c/asm_linkage.h", line 107: warning: comment is replaced by "##"

Fix:

Replace /****/ with ##

/usr/include/sun4c/debug/asm_linkage.h

Problem:

"/usr/include/sun4c/debug/asm_linkage.h", line 45: warning: comment is replaced by "##"

Fix:

Replace /****/ with ##

/usr/include/sundev/scsi.h

Problem:

"/usr/include/sundev/scsi.h", line 254: warning: syntax requires ";" after last struct/union member

"/usr/include/sundev/scsi.h", line 254: warning: unnamed struct member

Fix:

Add name followed by semicolon for the structure member

/usr/include/sunif/if_llc.h

Problem:

"/usr/include/sunif/if_llc.h", line 54: warning: syntax requires ";" after last struct/union member

Fix:

Add semicolon

/usr/include/suntool/wmgr.h

Problem:

"/usr/include/suntool/wmgr.h", line 3: warning: tokens ignored at end of directive line

Fix:

Change the #ifdef statment from wmgr.h_DEFINED to wmgr_h_DEFINED

/usr/include/sunwindow/io_stream.h

Problem:

"/usr/include/sunwindow/io_stream.h", line 112: warning: old-style declaration; add "int"

"/usr/include/sunwindow/io_stream.h", line 158: warning: old-style declaration; add "int"

"/usr/include/sunwindow/io_stream.h", line 170: warning: old-style declaration; add "int"

Fix:

Add missing return type for function declaration

/usr/include/sunwindow/sun.h

Problem:

"/usr/include/sunwindow/sun.h", line 21: warning: identifier redeclared: sprintf

Fix:

Add proper declaration

/usr/include/sys/debug.h

Problem:

"/usr/include/sys/debug.h", line 23: warning: macro replacement within a string literal

Fix:

Replace /****/ with ##

/usr/include/sys/ioccom.h

Note – For all modes except -Xs, ALL macro calls that use _IO, _IOR, _IORN, _IOW, _IOWN, _IOWR, and IOWRN must be changed. For example::

```
#ifdef __STDC__
          int i = _IO('z');
#else
          int i = _IO(z);
#endif
```

Problem:

"usr/include/sys/ioccom.h", line 25: warning: macro replacement within a character constant

"/usr/include/sys/ioccom.h", line 26: warning: macro replacement within a character constant

"/usr/include/sys/ioccom.h", line 27: warning: macro replacement within a character constant

"/usr/include//sys/ioccom.h", line 28: warning: macro replacement within a character constant

"/usr/include/sys/ioccom.h", line 29: warning: macro replacement within a character constant

"/usr/include/sys/ioccom.h", line 31: warning: macro replacement within a character constant

"/usr/include/sys/ioccom.h", line 32: warning: macro replacement within a character constant

The following files use macros from sys/ioccom.h and have been modified to work with the changed macros in sys/ioccom.h

/usr/include/net/nit_buf.h
/usr/include/net/nit_if.h
/usr/include/net/nit_pf.h
/usr/include/pixrect/gp1var.h
/usr/include/sbusdev/audio_79C30.h
/usr/include/sun/audioio.h
/usr/include/sun/dkio.h
/usr/include/sun/fbio.h
/usr/include/sun/gpio.h
/usr/include/sun/mem.h
/usr/include/sun/ndio.h
/usr/include/sun/sqz.h
/usr/include/sun/tvio.h
/usr/include/sun/vddrv.h
/usr/include/sundev/dbio.h
/usr/include/sundev/fdreg.h
/usr/include/sundev/kbio.h
/usr/include/sundev/mcpcmd.h
/usr/include/sundev/msio.h
/usr/include/sundev/msreg.h
/usr/include/sundev/openpromio.h
/usr/include/sundev/ppreg.h
/usr/include/sundev/srreg.h
/usr/include/sundev/streg.h

/usr/include/sundev/vuid_event.h
/usr/include/sunwindow/win_ioctl.h
/usr/include/sys/des.h
/usr/include/sys/filio.h
/usr/include/sys/mtio.h
/usr/include/sys/sockio.h
/usr/include/sys/stropts.h
/usr/include/sys/termio.h
/usr/include/sys/termios.h
/usr/include/sys/ttold.h
/usr/include/sys/ttycom.h
/usr/include/sys/vcmd.h
/usr/include/scsi/impl/uscsi.h
/usr/include/scsi/targets/srdef.h
/usr/include/scsi/targets/stdef.h

Fix:

Replace macros:

```
#ifdef __STDC__
#define _IO(x,y)        (_IOC_VOID | (x<<8) | y)
#define _IOR(x,y,t)     (_IOC_OUT | ((sizeof(t)&_IOCPARM_MASK)<<16) | (x<<8) | y)
#define _IORN(x,y,t)    (_IOC_OUT | (((t)&_IOCPARM_MASK)<<16) | (x<<8) | y)
#define _IOW(x,y,t)     (_IOC_IN | ((sizeof(t)&_IOCPARM_MASK)<<16) | (x<<8) | y)
#define _IOWN(x,y,t)    (_IOC_IN | (((t)&_IOCPARM_MASK)<<16) | (x<<8) | y)
#define _IOWR(x,y,t)    (_IOC_INOUT | ((sizeof(t)&_IOCPARM_MASK)<<16) | (x<<8) | y)
#define _IOWRN(x,y,t)   (_IOC_INOUT | (((t)&_IOCPARM_MASK)<<16) | (x<<8) | y)
#else
#define _IO(x,y)        (_IOC_VOID | ('x'<<8) | y)
#define _IOR(x,y,t)     (_IOC_OUT | ((sizeof(t)&_IOCPARM_MASK)<<16) | ('x'<<8) | y)
#define _IORN(x,y,t)    (_IOC_OUT | (((t)&_IOCPARM_MASK)<<16) | ('x'<<8) | y)
#define _IOW(x,y,t)     (_IOC_IN | ((sizeof(t)&_IOCPARM_MASK)<<16) | ('x'<<8) | y)
#define _IOWN(x,y,t)    (_IOC_IN | (((t)&_IOCPARM_MASK)<<16) | ('x'<<8) | y)
#define _IOWR(x,y,t)    (_IOC_INOUT | ((sizeof(t)&_IOCPARM_MASK)<<16) | ('x'<<8) | y)
#define _IOWRN(x,y,t)   (_IOC_INOUT | (((t)&_IOCPARM_MASK)<<16) | ('x'<<8) | y)
#endif
```

/usr/include/sys/termios.h

Note – For all modes except -Xs, ALL macro calls that use _CTRL must be changed. For example::

```
#ifdef __STDC__
        char ctrlz = _CTRL('z');
#else
        char ctrlz = _CTRL(z);
#endif
```

Problem:

"/usr/include/sys/termios.h", line 44: warning: macro replacement within a character constant

Fix:

Replace the macro _CTRL:

```
#ifdef __STDC__
#define _CTRL(c)        (c&037)
#else
#define _CTRL(c)        ('c'&037)
#endif
```

/usr/include/sys/ttychars.h

Note – For all modes except -Xs, ALL macro calls that use CTRL must be changed. For example:

```
#ifdef __STDC__
        char ctrlz = CTRL('z');
#else
        char ctrlz = CTRL(z);
#endif
```

Problem:

"/usr/include/sys/ttychars.h", line 29: warning: macro replacement within a character constant

Fix:

Replace the macro CTRL:

```
#ifdef __STDC__
#define CTRL(c) (c&037)
#else
#define CTRL(c) ('c'&037)
#endif
```

/usr/include/pixrect/pixrect.h

Problem:

Ending statements after an #endif of #else are not enclosed within comments

```
#endif KERNEL
```

Fix:

Place statment within comments

```
#endif /* KERNEL */
```

/usr/include/rpc/auth.h

Problem:

Statements are guarded with #ifdef sparc. sparc is not defined in -Xc mode.

Fix:

Remove #ifdef sparc

/usr/include/arpa/nameser.h

Problem:

Statements are guarded with #ifdef sparc. sparc is not defined in -Xc mode.

Fix:

Remove #ifdef sparc

/usr/include/sys/types.h

Problem:

Statements are guarded with #ifdef sparc. sparc is not defined in -Xc mode.

Fix:

Remove #ifdef sparc

/usr/include/sys/wait.h

Problem:

Statements are guarded with #ifdef sparc. sparc is not defined in -Xc mode.

Fix:

Remove #ifdef sparc

C.5 Problems with Header Files using the -Xc Mode

The following problems still occur with header files using the -Xc mode:

Bit fields which are not of type int or unsigned int

Problem:

"/usr/include/pixrect/gp1reg.h", line 39: warning: nonportable bit-field type

Tokens at the end of #else or #endif are not enclosed within comments

Problem:

"/usr/include/stand/sireg.h", line 274: warning: tokens ignored at end of directive line

Enumerated types which have a trailing comma

Problem:

/usr/include/sparc/fpu/ieee.h", line 46: warning: trailing "," prohibited in enum declaration

C.6 *Miscellaneous Differences*

C 2.0 (ANSI C) does not support the `#pragma weak` directive

Type Qualifier `const`

The type qualifier const is not placed in a read-only area of storage.

`size_t` *Type*

`size_t` is defined to be of type `int`, instead of `unsigned int` as specified by the ANSI standard.

-Xs Differences for Sun C and ANSI C

D.1 Introduction

In this appendix we describe the differences in compiler behavior when using the -Xs option. The -Xs option tries to emulate /bin/cc, Sun C 1.0, Sun C 1.1 (K&R style), but in some cases the emulation fails.

Table D-1 -Xs Behavior (Sheet 1 of 2)

data type	Sun C (K&R)	Sun ANSI C

aggregate initialization
 	
`struct {`
`  int a[3];`
`  int b;`
`} w[] = { {1} , 2};`
	`sizeof (w) = 16`
	`w[0].a = 1, 0, 0`
	`w[0].b =2`
		`sizeof (w) = 32`
		`w[0].a = 1, 0, 0`
		`w[0].b =2`

incomplete struct, union, enum declaration	`struct fq {`
	`  int i;`
	`  struct unknown;`
	`};`
		Does not allow incomplete struct, union, enum declaration.

switch expression integral type	Allows non-integral type.	Does not allow non-integral type.
order of precedence	Allows `if (rcount > count += index)`	Does not allow `if (rcount > count += index)`

Table D-1 -Xs Behavior (Sheet 2 of 2)

data type	Sun C (K&R)	Sun ANSI C

unsigned, short, and long typedef declarations	Allows
	
	`typedef short small`
	`unsigned small;`
		Does not allow (all modes).

struct or union tag mismatch in nested struct or union declarations	Allows tag mismatch
	
	`struct x {`
	`  int i;`
	`} s1;`
	
	`/* K&R treats as a struct */`
	`{`
	`  union x s2;`
	`}`
		Does not allow tag mismatch in nested struct or union declaration.

incomplete struct or union type	Ignores an incomplete type declaration.	`struct x {`
		`  int i;`
		`} s1;`
		
		`main()`
		`{`
		`  struct x;`
		`  struct y {`
		`  struct x f1;`
		`  /* in K&R, f1 refers */`
		`  /* to outer struct */`
		`  } s2;`
		`  struct x {`
		`  int i;`
		`  };`
		`}`

casts as lvalues	Allows `(char *) ip = &foo;`	Does not allow casts as lvalues (all modes).

Glossary

ANSI
ANSI is an acronym for the American National Standards Institute. ANSI establishes standards in the computing industry from the definition of ASCII (see below) to the measurement of overall datacom system performance. ANSI standards have been established for the Ada, FORTRAN, and C programming languages.

a.out
`a.out`, historically for "assembler output," is the default file name for an executable program produced by the C compilation system.

application
An application program is a working program in a given operating system, that is, an application of that system. When the source code for an application program is portable to another operating system, the program is an application of that system as well.

archive
An archive, or statically linked library, is a collection of object files each of which contains the code for a function or a group of related functions in the library. When you call a library function in your program and specify a static linking option on the `cc` command line, a copy of the object file that contains the function is incorporated in your executable at link time. For a discussion, see the *C 2.0 Libraries Reference Manual*.

argument

You use an argument to pass information to a command or a function. A command instructs the operating system to execute a program. The command is the name of the file containing the program. Command line arguments are character strings or numbers that follow the command, separated from it by a space, or that follow another command line argument, separated from it by a space. There are two types of command line arguments: options and operands. Options, which are immediately preceded by a minus sign (–), change the behavior of the program. Some options can themselves take arguments. Options are also called flags. Operands specify files or directories to be operated on by the program. So, in the command line % cc –o hello hello.c all the elements after the `cc` command are arguments. `cc` is the name of the file containing the C compiler program. The C source file `hello.c` is its operand. `-o` is an option that tells the compilation system to generate an executable program with a name other than `a.out`. `hello` is an argument to `-o` that specifies the name of the executable program to be created.

Function arguments are enclosed in a pair of parentheses immediately following the function name. The number of arguments can be zero or more; if two or more are given, they must be separated by commas and the whole list enclosed by parentheses. The formal definition of a function describes the number and data type of arguments expected by the function.

You can find formal definitions of the functions supplied with the C compilation system in the *SunOS 5.0 Reference Manual*.

ASCII

ASCII is an acronym for the American Standard Code for Information Interchange, the standard for data representation followed in the UNIX system. ASCII code represents 128 upper- and lowercase letters, numerals, and special characters as binary numbers. Each alphanumeric and special character has an ASCII equivalent that is one byte long.

assembler

Assembly language is a programming language that uses symbolic names to represent the machine instructions of a given computer. An assembler is a program that accepts instructions written in the assembly language of the computer and translates them into a binary representation of the corresponding machine instructions. Because each assembly language

instruction usually has a one-to-one correspondence with a machine instruction, programs written in assembly language are not portable to different machines.

buffer
A buffer is a space in computer memory where data are stored temporarily in convenient units for system operations. Buffers are often used by programs such as editors that access and alter text or data frequently. When you edit a file, for instance, a copy of its contents are read into a buffer; the copy is what you change. For your changes to become part of the permanent file, you must write the buffer's contents back into the permanent file. This replaces the contents of the file with the contents of the buffer. When you quit the editor, the contents of the buffer are flushed.

child process
See "`fork()`."

command
A command instructs the operating system to execute a program. On the UNIX system, an executable program is a compiled and linked program or a shell program. The command to execute a program is either is the name of the file containing the program. A command line consists of the command followed by its arguments, so % cc file1.c file2.c instructs the operating system to execute the C compiler program, which is stored in the file `cc`, and to use the source files `file1.c` and `file2.c` as input. A command line can extend over multiple terminal lines.

compiler
A compiler is a program that translates a source program written in a higher-level language into the assembly language of the computer the program is to run on. An assembler translates the assembly language code into the machine instructions of the computer. On the C compilation system, these instructions are stored in object files that correspond to each of your source files. That is, each object file contains a binary representation of the C language code in the corresponding source file. Source file names must end with the characters `.c`; object files take the name of the source file with `.o` in place of `.c`. The link editor links these object files with each other, and with any library functions you have used in your source code, to produce an executable program called `a.out` by default. The preprocessor component of the C compiler performs macro expansion, conditional compilation, and file inclusion before the compiler proper translates C source code into assembly language.

core image
A core image is a copy of the memory image of a process. A file named `core` is created in your current directory when the UNIX system aborts an executing program. The file contains the core image of the process at the time of the failure.

data symbol
A data symbol names a variable that may or may not be initialized. Normally, these variables reside in read/write memory during execution. Compare "text symbol."

debugging
Debugging is the process of locating and correcting errors in executable programs.

default
A default is the way a program will perform a task in the absence of other instructions, that is, if you do not specify something else.

directory
A directory is a type of file used to group and organize other files or directories. A subdirectory is a directory that is pointed to by a directory one level above it in the file system. A directory name is a string of characters that identifies the directory. It can be a simple directory name, a relative path name, or a full path name.

dynamic linking
Dynamic linking refers to the process in which external references in a program are linked with their definitions when the program is executed. For a discussion, see the *C 2.0 Libraries Reference Manual* and the *Linker and Libraries Manual SunOS 5.0.*

ELF
ELF is an acronym for the executable and linking format of the object files produced by the C compilation system.

environment
An environment is a collection of resources used to support a function. On the UNIX system, the shell environment consists of variables whose values define the way you interact with the operating system. The shell environment variable `$HOME`, for example, stands for your login directory; `$PATH` is a list of directories the shell will search for executable programs. When you log in, the system executes programs that create most of the

environment variables you need to do your work. These variables are stored in `/etc/profile`, a file that defines a common environment for users when they log in to the system. You can tailor your environment to your own needs by defining and setting variables in the file `.profile` (or `.cshrc` or `.login` in a C Shell) in your login directory. You can also temporarily set variables at the shell level.

executable program
On the UNIX system, an executable program is a compiled and linked program or a shell program. The command to execute either is the name of the file containing the program. A compiled and linked program is called an executable object file. Compare "object file."

exit()
The `exit()` function causes a process to terminate. `exit()` closes any open files and cleans up most other information and memory used by the process. An exit status, or return code, is an integer value that your program returns to the operating system to say whether it completed successfully or not.

expression
An expression is a mathematical or logical symbol or meaningful combination of symbols.

file
A file is a potential source of input or a potential destination for output; at some point, then, an identifiable collection of information. A file is known to the UNIX system as an inode plus the information the inode contains that tells whether the file is a plain file, a special file, or a directory. A plain file contains text, data, programs, or other information that forms a coherent unit. A special file is a hardware device or portion thereof, such as a disk partition. A directory is a type of file that contains the names and inode addresses of other plain, special, or directory files.

file descriptor
A file descriptor is an integer value assigned by the operating system to a file when the file is opened by a process.

file system
A UNIX file system is a hierarchical collection of directories and other files that are organized in a tree structure. The base of the structure is the root (/) directory; other directories, all subordinate to root, are branches. The

collection of files can be mounted on a block special file. Each file of a file system appears exactly once in the inode list of the file system and is accessible via a single, unique path from the root directory of the file system.

filter
A filter is a program that reads information from the standard input, acts on it in some way, and sends its result to the standard output. It is called a filter because it can be used in a pipeline (see "pipe") to transform the output of another program. Filters are different from editors in that they do not change the contents of a file. Examples of UNIX system filters are `sort`, which sorts the input, and `wc`, which counts the number of words, characters, and lines in the input.

flag
See "argument."

fork()
`fork()` is a system call that splits one process into two, the parent process and the child process, with separate, but initially identical, text, data, and stack segments. `fork()` is described in Section 2 of the *SunOS Reference Manual.*

header file
A header file is a file that usually contains shared data declarations that are to be copied into source files by the compiler. Header file names conventionally end with the characters `.h`. Header files are also called include files, for the C language `#include` directive by which they are made available to source files.

include file
See "header file."

interrupt
An interrupt is a break in the normal flow of a system or program. Interrupts are initiated by signals generated by a hardware condition or a peripheral device to indicate the occurrence of a specified event. When the interrupt is recognized by the hardware, an interrupt handling routine is executed. An interrupt character is a character (normally ASCII) that, when typed on a terminal, causes an interrupt. You can usually interrupt UNIX system programs by pressing the `delete` or `break` keys, or by pressing the `CTRL` and `d` keys simultaneously.

I/O

I/O stands for input/output, the process by which information enters (input) and leaves (output) a computer system.

kernel

The kernel is the basic resident software of the UNIX system. The kernel is responsible for most system operations: scheduling and managing the work done by the computer, maintaining the file system, and so forth. The kernel has its own text, data, and stack areas.

lexical analysis

Lexical analysis is the process by which a stream of characters (often comprising a source program) is broken up into its elementary words and symbols, called tokens. The tokens can include the reserved words of a programming language, its identifiers and constants, and special symbols such as `=`, `:=`, and `;`. Lexical analysis enables you to recognize, for instance, that the stream of characters `printf("hello, world\n");` is a series of tokens beginning with `printf` and not with, say, `printf("h`. In compilers, a lexical analyzer is often called by a syntactic analyzer, or parser, that analyzes the grammatical form of tokens passed to it by the lexical analyzer.

library

A library is a file that contains object code for a group of commonly used functions. Rather than write the functions yourself, you arrange for the functions to be linked with your program when an executable is created (see "archive") or when it is run (see "shared object").

link editing

Link editing refers to the process in which a symbol referenced in one module of a program is connected with its definition in another. On the C compilation system, programs are linked statically, when an executable is created, or dynamically, when it is run. See the *Linker and Libraries Manual SunOS 5.0* and the *C 2.0 Libraries Reference Manual.*

makefile

A `makefile` is a file that is used with the program `make` to keep track of the dependencies between modules of a program, so that when one module is changed, dependent ones are brought up to date.

module
A module is a program component that typically contains a function or a group of related functions. Source files and libraries are modules.

null pointer
A null pointer is a C pointer with a value of 0.

object file
An object file contains a binary representation of programming language code. A relocatable object file contains references to symbols that have not yet been linked with their definitions. An executable object file is a linked program. Compare "source file."

optimizer
An optimizer improves the efficiency of the assembly language code generated by a compiler. That, in turn, will speed the execution time of your object code.

option
See "argument."

parent process
See "`fork()`."

parser
A parser, or syntactic analyzer, analyzes the grammatical form of tokens passed to it by a lexical analyzer (see "lexical analysis").

path name
A path name designates the location of a file in the file system. It is made up of a series of directory names that proceed down the hierarchical path of the file system. The directory names are separated by a slash character (/). The last name in the path is the file. If the path name begins with a slash, it is called an absolute, or full, path name; the initial slash means that the path begins at the root directory. A path name that does not begin with a slash is known as a relative path name, meaning relative to your current directory.

permissions
Permissions define a right to access a file in the file system. Permissions are granted separately to you, your group, and all others. There are three basic permissions: read, write, and execute.

pipe
A pipe causes the output of one program to be used as the input to another program, so that the programs run in sequence. You create a pipeline by preceding each command after the first command with the pipe symbol (|), which indicates that the output from the process on the left should be routed to the process on the right. So % who | wc –l causes the output of the `who` command, which lists the users who are logged in to the system, to be used as the input of the `wc`, or word count, command with the `-l` option. The result is the number of users logged in to the system.

portability
Portability refers to the degree of ease with which a program can be moved to a different operating system or machine and run.

preprocessor
A preprocessor is a a program that prepares an input file for another program. The preprocessor component of the C compiler performs macro expansion, conditional compilation, and file inclusion.

process
A process is an executing program. Every time you enter the name of a file that contains an executable program you initiate a new process. A process ID is a unique system-wide number that identifies an active process. You can use the `ps`(1) command to determine the process ID of any process currently active on your system.

regular expression
A regular expression is a string of alphanumeric characters and special characters that describes, in a shorthand way, a pattern to be searched for in a file.

routine
A routine is another name for a function.

shared object
A shared object, or dynamically linked library, is a single object file that contains the code for every function in the library. When you call a library function in your program, and specify a dynamic linking option on the `cc` command line, the entire contents of the shared object are mapped into the virtual address space of your process at run time. As its name implies, a

shared object contains code that can be used simultaneously by different programs at run time. For a discussion, see the *C 2.0 Libraries Reference Manual* and the *Linker and Libraries Manual SunOS 5.0*.

shell
The shell is the UNIX system program that handles communication between you and the system. The shell is known as a command interpreter because it translates your commands into a language understandable by the system. A shell normally is started for you when you log in to the system. A shell program calls the shell to read and execute commands contained in an executable file.

signal
A signal is a message you send to a process or that processes send to one another. You might use a signal, for example, to initiate an interrupt. A signal sent by a running process is usually a sign of an exceptional occurrence that has caused the process to terminate or divert from the normal flow of control.

source file
Source files contain the programming language version of a program. Before a computer can execute the program, the source code must be translated by a compiler and assembler into the machine language of the computer. Compare "object file."

standard error
Standard error is an output stream from a program that normally is used to convey error messages. On the UNIX system, the default case is to associate standard error with the user's terminal.

standard input
Standard input is an input stream to a program. On the UNIX system, the default case is to associate standard input with the user's terminal.

standard output
Standard output is an output stream from a program. On the UNIX system, the default case is to associate standard output with the user's terminal.

static linking
Static linking refers to the process in which external references in a program are linked with their definitions when an executable is created. For a discussion, see the *C 2.0 Libraries Reference Manual*.

stream
A stream is an open file with its associated buffering. *Stream* also refers to a full duplex processing and data transfer path in the kernel that implements a connection between a driver in kernel space and a process in user space, providing a general input/output interface for user processes.

string
A string is a contiguous sequence of characters treated as a unit. In C, a character string is an array of characters terminated by the null character, `\0`.

syntax
Command syntax is the order in which commands and their arguments must be put together. The command always comes first. The order of arguments varies from command to command. Language syntax is the set of rules that describes how the elements of a programming language may legally be used.

system call
A system call is a request from a program for an action to be performed by the UNIX system kernel.

text symbol
A text symbol names a program instruction. Instructions reside in read-only memory during execution. Compare "data symbol."

user ID
A user ID is an integer value, usually associated with a login name, that the system uses to identify owners of files and directories. The user ID of a process becomes the owner of files created by the process and by descendant processes (see "`fork()`").

variable
In a program, a variable is an object whose value may change during the execution of the program or from one execution to the next. A variable in the shell is a name representing a string of characters.

white space
White space is one or more spaces, tabs, or new-line characters. White space is normally used to separate strings of characters and is required to separate the command from its arguments on a command line.

Index

Symbols

A

B

C

D

E

F

G

M

N

O

P

Q

R

S

SunPro

A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunPro Europe
13, Avenue Morane Saulnier
78140 Velizy, France

800-6578-10