6

Running the compiler

In the previous chapter, we looked at building from the viewpoint of *make*. The other central program in the build process is the compiler, which in UNIX is almost always a C compiler. Like *make*, the compiler can discover a surprising number of problems in what ostensibly debugged source code. In this chapter, we'll look at these problems and how to solve them. In we'll look at how the compiler works and how the various flavours of C differ. Although we restrict our attention to the C compiler, much of what we discuss relates to other compilers as well, particularly of course to C++. This chapter expects a certain understanding of the C language, of course, but don't be put of if you're still a beginner: this is more about living with C than writing it.

Information from the compiler can come in a number of forms:

- The compiler may issue *warnings*, which are informational messages intended to draw attention to possible program errors. Their reliability and their value varies significantly: some are a sure-fire indication that something is wrong, while others should be taken with a pinch of salt.
- The compiler may issue *error* messages, indicating its conviction that it cannot produce a valid output module. This also usually means that the compiler will not create any output files, though you can't always rely on this.
- The compiler may fail completely, either because of an internal bug or because it realizes that it no longer understands the input sufficiently to continue.

Compiler warnings

It's easy to make mistakes when writing programs, but it used to be even easier: nowadays, even the worst compilers attempt to catch dubious constructs and warn you about them. In this section, we'll look at what they can and can't do.

Before compilers worried about coding quality, the program *lint* performed this task. *lint* is still around, but hardly anybody uses it any more, since it doesn't always match the compiler being used. This is a pity, because *lint* can catch a number of dubious situations that evade most compilers.

Modern compilers can recognize two kinds of potential problems:

- Problems related to dubious program text, like

  ```
  if (a = 1)
    return;
  ```

  The first line of this example is almost superfluous: if I allocate the value `1` to `a`, I don't need an `if` to tell me what the result will be. This is probably a typo, and the text should have been

  ```
  if (a == 1)
    return;
  ```

- Problems related to program flow. These are detected by the flow analysis pass of the optimizer. For example:

  ```
  int a;
  b = a;
  ```

  The second line uses the value of `a` before it has been assigned a value. The optimizer notices this omission and may print a warning.

In the following sections, we'll examine typical warning messages, how they are detected and how reliable they are. I'll base the sections on the warning messages from the GNU C compiler, since the it has a particularly large choice of warning messages, and since it is also widely used. Other compilers will warn about the same kind of problems, but the messages may be different. Table 6-1 gives an overview of the warnings we'll see.

Table 6−1: Overview of warning messages

Table 6−1: Overview of warning messages (continued)

Kind of warning	page
Changing non-volatile automatic variables	82
Character subscripts to arrays	80
Dequalifying types	81
Functions with embedded extern definitions	84
Implicit conversions between enums	82
Implicit return type	79
Incomplete switch statements	82
Inconsistent function returns	79
Increasing alignment requirements	81
Invalid keyword sequences in declarations	83
Long indices for switch	82
Missing parentheses	83
Nested comments	83
Signed comparisons of unsigned values	80
Trigraphs	83
Uninitialized variables	80

Implicit return type

K&R C allowed programs like

```
main ()
  {
  printf ("Hello, World!\n");
  }
```

ANSI C has two problems with this program:

- The function name `main` does not specify a return type. It defaults to `int`.
- Since `main` is implicitly an `int` function, it should return a value. This one does not.

Both of these situations can be caught by specifying the `-Wreturn-type` option to *gcc*. This causes the following messages:

```
$ gcc -c hello.c -Wreturn-type
hello.c:2: warning: return-type defaults to ‘int’
hello.c: In function ‘main’:
hello.c:4: warning: control reaches end of non-void function
```

Inconsistent function returns

The following function does not always return a defi ned value:

```
foo (int x)
  {
  if (x > 3)
    return x - 1;
  }
```

If `x` is greater than 3, this function returns `x - 1`. Otherwise it returns with some uninitialized value, since there is no explicit `return` statement for this case. This problem is particularly insidious, since the return value will be the same for every invocation on a particular architecture (possibly the value of `x`), but this is a by-product of the way the compiler works, and may be completely different if you compile it with a different compiler or on some other architecture.

Uninitialized variables

Consider the following code:

```
void foo (int x)
  {
  int a;
  if (x > 5)
    a = x - 3;
  bar (a);
  ... etc
```

Depending on the value of `x`, `a` may or may not be initialized when you call `bar`. If you select the `-Wuninitialized` compiler option, it warns you when this situation occurs. Some compilers, including current versions of *gcc* place some limitations on this test.

Signed comparisons of unsigned values

Occasionally you see code of the form

```
int foo (unsigned x)
  {
  if (x >= 0)
...  etc
```

Since `x` is unsigned, its value is *always* >= 0, so the `if` is superfluous. This kind of problem is surprisingly common: system header files may differ in opinion as to whether a value is signed or unsigned. The option `-W` causes the compiler to issue warnings for this and a whole lot of other situations.

Character subscripts to arrays

Frequently, the subscript to an array is a character. Consider the following code:

```
char iso_translate [256] =    /* translate table for ISO 8859-1 to LaserJet */
{
 codes for the first 160 characters
```

```
  0xa0, 0xa1, 0xa2, 0xa3, 0xa4, 0xa5, 0xa6, 0xa7,
  0xa8, 0xa9, 0xaa, 0xab, 0xac, 0xad, 0xae, 0xaf,
...  etc
  };

#define xlate(x) iso_translate [x];

char *s;                      /* pointer in buf */
for (*s = buf; *s; s++)
  *s = xlate (*s);
```

The intention of `xlate` is to translate text to a form used by older model HP LaserJet printers. This code works only if the `char *s` is unsigned. By default, the C `char` type is a signed value, and so the characters `0x80` to `0xff` represent a negative array offset, and the program attempts (maybe successfully) to access a byte outside the table `iso_translate`. *gcc* warns about this if you set the option `-Wchar-subscripts`.

Dequalifying types

The following code fragment can cause problems:

```
char *profane;
void foo (const char *holy)
  {
  profane = holy;
```

The assignment of `holy` to `profane` loses the qualifier `const`, and the compiler complains about the fact. On the other hand, this is valid:

```
  profane = (char *) holy;
```

This doesn't make it a better idea: `holy` is supposed to be unchangeable, and here you are removing this qualifier. If you specify the `-Wcast-qual` option to *gcc*, it complains if you use a cast to remove a type qualifier such as *const*.

Increasing alignment requirements

Many processors require that specific data types be aligned on specific boundaries, and the results can be spectacular if they are not—see Chapter 11, *Hardware dependencies*, page 158, for more details. We can easily outsmart the C compiler with code like:

```
void foo (char *x)
  {
  int *ip = (int *) x;
```

In this case, there is a good chance that the `int *` pointer `ip` requires a specific alignment and is not allowed to point at any address in memory the way the `char` pointer `x` is allowed to do. If you specify the `-Wcast-align` option to *gcc*, it warns you of such assignments.

Implicit conversions between enums

One of the advantages of *enums* is that they make type checking easier—we'll look at that in more detail in Chapter 20, *Compilers*, page 339. If you specify the `-Wenum-clash` option to *gcc*, *and* you're compiling C++, it warns about sloppy use of *enum*s.

Incomplete switch statements

A frequent cause of error in a *switch* statement is that the index variable (the variable that decides which case is chosen) may assume a value for which no *case* has been specified. If the index variable is an `int` of some kind, there is not much you can do except include a `default` clause. If the index variable is an `enum`, the compiler can check that `case` clauses exist for all the possible values of the variable, and warns if they do not. It also warns if `case` clauses exist for values that are not defined for the type of the index variable. Specify the `-Wswitch` option for these warnings.

long indices for switch

In some dialects of pre-ANSI C, you could write things like

```
foo (x)
long x;
  {
  switch (x)
    {
...  etc
```

This is no longer allowed in ANSI C: indices for `switch` must evaluate to an `int`, even if `int` and `long` have the same length. *gcc* issues a warning about `long` indices in `switch` unless you specify the `-traditional` option.

Changing non-volatile automatic variables

Under certain circumstances, a signal handler might modify a local automatic variable if the function has called `setjmp`—see Chapter 13, *Signals*, page 200 for more details. *gcc* options this situation as a warning if you specify the `-W` option. This is a complicated problem:

- It can occur only during an optimizing compilation, since the keyword *volatile* has meaning only in these circumstances. In addition, the situation is recognized only by the optimizer.
- The optimizer cannot recognize when a `longjmp` could be performed. This depends on semantics outside the scope of the optimizer. As a result, it could issue this warning when there is, in fact, no danger.

Invalid keyword sequences in declarations

Currently, it is permissible to write declarations like

```
int static bad_usage;
```

Here the storage class specifier *static* comes after the type specifier `int`. The ANSI Standard still permits this, but declares the usage to be obsolescent. *gcc* issues a warning when it encounters this and the option `-W` has been set.

Trigraphs

Trigraphs (see Chapter 20, *Compilers*, page 342) are no error, at least according to the ANSI Standard. The Free Software Foundation makes no bones about their opinion of them, and so *gcc* supplies the option `-Wtrigraphs`, which prints a warning if any trigraphs occur in the source code. Since this works only if the option `-trigraphs` is used to enable them, it is not clear that this is of any real use.

Nested comments

Occasionally you see code like

```
void foo (int x)
  {
  int y;                    /* state information
  y = bar ();               /* initialize y */
  if (y == 4)
...  etc
```

The code looks reasonable, and it is syntactically correct C, but in fact the comment after the declaration of `y` is not terminated, so it includes the whole of the next line, which is almost certainly not the intention. *gcc* recognizes this if it finds the sequence `/*` in a comment, and warns of this situation if you specify the `-Wcomment` option.

Missing parentheses

What value does the following code return?

```
int a = 11 << 4 & 7 << 2 > 4;
```

The result is 0, but the real question is: in what order does the compiler evaluate the expression? You can find the real answer on page 53 of K&R, but you don't want to do that all the time. We can re-write the code as

```
int a = (11 << 4) & ((7 << 2) > 4);
```

This makes it a lot clearer what is intended. *gcc* warns about what it considers to be missing parentheses if you select the `-Wparentheses` option. By its nature, this option is subjective, and you may find that it complains about things that look fine to you.

Functions with embedded extern definitions

K&R C allowed you to write things like

```
int datafile;
foo (x)
  {
  extern open ();
  datafile = open ("foo", 0777);
  }
```

The *extern* declaration was then valid until the end of the source file. In ANSI C, the scope of `open` would be the scope of `foo`: outside of `foo`, it would no longer be known. *gcc* issues a warning about `extern` statements inside a function definition unless you supply the `-traditional` option. If you are using `-traditional` and want these messages, you can supply the `-Wnested-externs` option as well.

Compiler errors

Of course, apart from warnings, you frequently see error messages from the compiler—they are the most common reason for a build to fail. In this section, we'll look at some of the more common ones.

Undefined symbols

This is one of the most frequent compiler error messages you see during porting. At first sight, it seems strange that the compiler should find undefined symbols in a program that has already been installed on another platform: if there are such primitive errors in it, how could it have worked?

In almost every case, you will find one of the following problems:

- The definition you need may have been `#ifdef`'ed out. For example, in a manually configured package, if you forget to specify a processor architecture, the package may try to compile with no processor definitions, which is sure to give rise to this kind of problem.
- The symbol may have been defined in a header file on the system where it was developed. This header file is different on your system, and the symbol you need is never defined.
- You may be looking at the wrong header files. Some versions of *gcc* install "fixed" copies of the system header files in their own private directory. For example, under BSD/386 version 1.1, *gcc* version 2.6.3 creates a version of *unistd.h* and hides it in a private directory. This file omits a number of definitions supplied in the BSDI version of *unistd.h*. You can confirm which header files have been included by running *gcc* with the `-H` option. In addition, on page 86 we look at a way to check exactly what the preprocessor did.

The second problem is surprisingly common, even on supposedly identical systems. For

example, in most versions of UNIX System V.4.2, the system header file *link.h* defines information and structures used by debuggers. In UnixWare 1.0, it defines information used by some Novell-specific communications protocols. If you try to compile *gdb* under UnixWare 1.0, you will have problems as a result: the system simply does not contain the definitions you need.

Something similar happens on newer System V systems with POSIX.1 compatibility. A program that seems formally correct may fail to compile with an undefined symbol `O_NDELAY`. `O_NDELAY` is a flag to `open`, which specifies that the call to `open` should not wait for completion of the request. This can be very useful, for example, when the `open` is on a serial line and will not complete until an incoming call occurs. The flag is supported by almost all modern UNIX ports, but it is not defined in POSIX.1. The result is that the definition is carefully removed if you compile defining `-D_POSIX_SOURCE`.

You might think that this isn't a problem, and that you can replace `O_NDELAY` with the POSIX.1 flag `O_NONBLOCK`. Unfortunately, the semantics of `O_NONBLOCK` vary from those of `O_NDELAY`: if no data is available, `O_NONBLOCK` returns -1, and `O_NDELAY` returns 0. You can make the change, of course, but this requires more modifications to the program, and you have a strraighforward alternative: `#undef _POSIX_SOURCE`. If you do this, you may find that suddenly other macros are undefined, for example `O_NOCTTY`. System V.4 only defines this variable if `_POSIX_SOURCE` is set.

There's no simple solution to this problem. It is caused by messy programming style: the programmer has mixed symbols defined only by POSIX.1 with those that are not defined in POSIX.1. The program may run on your current system, but may stop doing so at the next release.

Conflicts between preprocessor and compiler variables

Occasionally you'll see things that seem to make absolutely no sense at all. For example, porting *gcc*, I once ran into this problem:

```
gcc -c  -DIN_GCC   -g -O3     -I.  -I.  -I./config \
          -DGCC_INCLUDE_DIR=\"/opt/lib/gcc-lib/i386--sysv/2.6.0/include\" \
  -DGPLUSPLUS_INCLUDE_DIR=\"/opt/lib/g++-include\" \
          -DCROSS_INCLUDE_DIR=\"/opt/lib/gcc-lib/i386--sysv/2.6.0/sys-include\" \
  -DTOOL_INCLUDE_DIR=\"/opt/i386--sysv/include\" \
  -DLOCAL_INCLUDE_DIR=\"/usr/local/include\" \
  -DSTD_PROTO_DIR=\"/opt/lib/gcc-lib/i386--sysv/2.6.0\" \
  ./protoize.c
./protoize.c:156: macro ‘puts’ used without args
```

Looking at this part of *protoize.c*, I found lots of external definitions:

```
extern int fflush ();
extern int atoi ();
extern int puts ();
extern int fputs ();
extern int fputc ();
extern int link ();
extern int unlink ();
```

Line 156 is, not surprisingly, the definition of `puts`. But this is a definition, not a call, and certainly not a macro. And why didn't it complain about all the other definitions? There were many more than shown here.

In cases like this, it's good to understand the way the compiler works—we'll look at this in more detail in Chapter 20, *Compilers*, on page 348. At the moment, we just need to recall that programs are compiled in two stages: first, the preprocessor expands all preprocessor definitions and macros, and then the compiler itself compiles the resultant output, which can look quite different.

If you encounter this kind of problem, there's a good chance that the compiler is not seeing what you expect it to see. You can frequently solve this kind of riddle by examining the view of the source that the compiler sees, the output of the preprocessor. In this section, we'll look at the technique I used to solve this particular problem.

All compilers will allow you to run the preprocessor separately from the compiler, usually by specifying the `-E` option—see your compiler documentation for more details. In this case, I was running the compiler in an *xterm*[*], so I was able to cut and paste the complete 8-line compiler invocation as a command to the shell, and all I needed to type was the text in bold face:

```
$ gcc -c  -DIN_GCC   -g -O3     -I.  -I.  -I./config \
          -DGCC_INCLUDE_DIR=\"/opt/lib/gcc-lib/i386--sysv/2.6.0/include\" \
  -DGPLUSPLUS_INCLUDE_DIR=\"/opt/lib/g++-include\" \
          -DCROSS_INCLUDE_DIR=\"/opt/lib/gcc-lib/i386--sysv/2.6.0/sys-include\" \
  -DTOOL_INCLUDE_DIR=\"/opt/i386--sysv/include\" \
  -DLOCAL_INCLUDE_DIR=\"/usr/local/include\" \
  -DSTD_PROTO_DIR=\"/opt/lib/gcc-lib/i386--sysv/2.6.0\" \
  ./protoize.c -E -o junk.c
$
```

If you don't have *xterm*, you can do the same sort of thing by editing the make log (see Chapter 5, *Building the package*, page 60), which will contain the invocation as well.

junk.c starts with:

```
# 1 "./config.h" 1


# 1 "./config/i386/xm-i386.h" 1
40 empty lines
# 1 "./tm.h" 1
19 empty lines
# 1 "./config/i386/gas.h" 1
22 empty lines
```

This file seems to consist mainly of empty lines, and the lines that aren't empty don't seem to be C! In fact, the # lines *are* C (see the `line` directive in Chapter 20, *Compilers*, page 344), except that in this case the keyword `line` has been omitted. The empty lines are where comments and preprocessor directives used to be. The error message referred to line 156 of *protoize.c*, so I searched for lines with `protoize.c` on them. I found a number of them:

* *xterm* is a terminal emulator program that runs under X11. If you don't use X11, you should—for example, it makes this particular technique much easier.

```
$ grep protoize.c junk.c
# 1 "./protoize.c"
# 39 "./protoize.c" 2
# 59 "./protoize.c" 2
# 62 "./protoize.c" 2
# 63 "./protoize.c" 2
...  etc
# 78 "./protoize.c" 2
# 222 "./protoize.c"
```

Clearly, the text was between lines 78 and 222. I positioned on the line *after* the marker for line 78 and moved down (156 - 78) or 78 lines. There I found:

```
extern int fflush ();
extern int atoi ();
extern int ((fputs(( ), stdout) || (( stdout )->__bufp < ( stdout )->__put_limit
? (int) (unsigned char) (*( stdout )->__bufp++ = (unsigned char) ( '0 ))
:__flshfp (( stdout ), (unsigned char) ( '0 ))) == (-1) ) ? (-1) : 0) ;
extern int fputs ();
extern int fputc ();
extern int link ();
extern int unlink ();
```

Well, at any rate this made it clear why the compiler was complaining. But where did this junk come from? It can be difficult to figure this out. With *gcc* you can use the `-dD` option to keep the preprocessor definitions—unfortunately, the compiler still removes the other preprocessor directives. I used `-dD` as well, and found in *junk.c*:

```
# 491 "/opt/include/stdio.h" 2
25 lines missing
extern int fputs  (__const char *__s, FILE *__stream)  ;
/* Write a string, followed by a newline, to stdout.  */
extern int puts  (__const char *__s)  ;

#define puts(s) ((fputs((s), stdout) || __putc('0, stdout) == EOF) ? EOF : 0)
```

This looks strange: first it declares `puts` as an external function, then it defines it as a macro. Looking at the original source of *stdio.h*, I found:

```
/* Write a string, followed by a newline, to stdout.  */
extern int puts __P ((__const char *__s));

#ifdef  __OPTIMIZE__
#define puts(s) ((fputs((s), stdout) || __putc('0, stdout) == EOF) ? EOF : 0)
#endif /* Optimizing.  */
```

No, this doesn't make sense—it's a real live bug in the header file. At the very least, the declaration of `puts ()` should have been in an `#else` clause. But that's not the real problem: it doesn't worry the preprocessor, and the compiler doesn't see it. The real problem is that *protoize.c* is trying to do the work of the header files and define `puts` again. There are many programs that try to out-guess header files: this kind of definition breaks them all.

There are at least two ways to fix this problem, both of them simple. The real question is, what is the Right Thing? System or library header files should be allowed to define macros

instead of functions if they want, and an application program has no business trying to do the work of the header files, so it would make sense to fix *protoize.c* by removing all these external definitions: apart from this problem, they're also incompatible with ANSI C, since they don't describe the parameters. In fact, I chose to remove the definition from the header file, since that way I only had to do the work once, and in any case, it's not clear that the definition really would run any faster.

Preprocessor output usually looks even more illegible than this, particularly if lots of clever nested `#define`s have been performed. In addition, you'll frequently see references to nonexistant line numbers. Here are a couple of ways to make it more legible:

- Use an editor to put comments around all the `#line` directives in the preprocessor output, and then recompile. This will make it easier to find the line in the preprocessor output to which the compiler or debugger is referring; then you can use the comments to follow it back to the original source.
- Run the preprocessor output through a program like *indent*, which improves legibility considerably. This is especially useful if you find yourself in the unenviable position of having to modify the generated sources. *indent* is not guaranteed to maintain the same number of lines, so after indenting you should recompile.

Other preprocessors

There are many other cases in which the source file you use is not the source file that the compiler gets. For example, *yacc* and *bison* take a grammar file and make a (more or less illegible) *.c* file out of it; other examples are database preprocessors like Informix ESQL, which takes C source with embedded SQL statements and converts it into a form that the C compiler can compile. The preprocessor's output is intended to be read by a compiler, not by humans.

All of these preprocessors use lines beginning with # to insert information about the original line numbers and source files into their output. Not all of them do it correctly: if the preprocessor inserts extra lines into the source, they can become ambiguous, and you can run into problems when using symbolic debuggers, where you normally specify code locations by line number.

Syntax errors

Syntax errors in previously functional programs usually have the same causes as undefined symbols, but they show their faces in a different way. A favourite one results from omitting */usr/include/sys/types.h*. For example, consider *bar.c*:

```
#include <stdio.h>
#ifdef USG
#include <sys/types.h>
#endif

ushort num;
int main (int argc, char *argv [])
{
```

```
num = atoi (argv [1]);
printf ("First argument: %d\n", num);
}
```

If you compile this under BSD/OS, you get:

```
$ gcc -o bar bar.c
bar.c:6: parse error before 'num'
bar.c:6: warning: data definition has no type or storage class
```

There's an error because `ushort` hasn't been defined. The compiler expected a type specifier, so it reported a syntax error, not an undefined symbol. To fix it, you need to define the type specified—see Appendix A, *Comparative reference to UNIX data types* for a list of the more common type specifiers.

Virtual memory exhausted

You occasionally see this message, particularly when you're using *gcc*, which has a particular hunger for memory. This may be due to unrealistically low virtual memory limits for your system—by default, some systems limit total virtual memory per process to 6 MB, but *gcc* frequently requires 16 or 20 MB of memory space, and on occasion it can use up to 32 MB for a single compilation. If your system has less than this available, increase the limit accordingly. Don't forget to ensure that you have enough swap space! Modern systems can require over 100 MB of swap space.

Sometimes this doesn't help. *gcc* seems to have particular difficulties with large data definitions; bit map definitions in X11 programs are the sort of things that cause problems. *xphoon*, which displays a picture of the current phase of the moon on the root window, is a good example of a *gcc*-breaker.

Compiler limits exceeded

Some compilers have difficulties with complicated expressions. This can cause *cc1*, the compiler itself, to fail with messages like "expression too complicated" or "out of tree space." Fixing such problems can be tricky. Straightforward code shouldn't give the compiler indigestion, but some nested *#define*s can cause remarkable increases in the complexity of expressions: in some cases, a single line can expand to over 16K of text. One way to get around the problem is to preprocess the code and then break the preprocessed code into simpler expressions. The *indent* program is invaluable here: preprocessor output is not intended to be human-readable, and most of the time it isn't.

Running compiler passes individually

Typical compilers run four distinct passes to compile and link a program—see Chapter 20, *Compilers*, page 348, for more details. Sometimes running the passes separately can be useful for debugging a compilation:

- If you find yourself with header files that confuse your preprocessor, you can run a different preprocessor, collect the output and feed it to your compiler. Since the output of the preprocessor is not machine-dependent, you could even do this on a different machine with different architecture, as long as you ensure that you use the correct system header files. By convention, the preprocessor output for *foo.c* would be called *foo.i*—see Chapter 20, *Compilers*, page 348 for a list of intermediate file suffixes—though it usually does no harm if you call it *foo.c* and pass it through the preprocessor again, since there should no longer be anything for the second preprocessor to do.
- If you want to report a compiler bug, it's frequently a good idea to supply the preprocessor output: the bug might be dependent on some header file conflict that doesn't exist on the system where the compiler development takes place.
- If you suspect the compiler of generating incorrect code, you can stop compilation after the compiler pass and collect the generated assembler output.

Incorrect code from compiler

Compilers sometimes generate incorrect code. Incorrect code is frequently difficult to debug because the source code looks (and might be) perfect. For example, a compiler might generate an instruction with an incorrect operand address, or it might assign two variables to a single location. About the only thing you can do here is to analyze the assembler output.

One kind of compiler bug is immediately apparent: if the code is so bad that the assembler can't assemble it, you get messages from the assembler. Unfortunately, the message doesn't usually tell you that it comes from the assembler, but the line numbers change between the compiler and the assembler. If the line number seems completely improbable, either because it is larger than the number of lines in your source file, or because it seems to have nothing to do with the context of that line, there is a chance that the assembler produced the message. There are various ways to confirm which pass of the compiler produced the message. If you're using *gcc*, the simplest one is to use the `-v` option for the compiler, which "announces" each pass of compilation as it starts, together with the version numbers and parameters passed to the pass. This makes it relatively easy to figure out which pass is printing the error messages. Otherwise you can run the passes individually—see Chapter 20, *Compilers*, page 348 for more details.