Compilers

The central tool in building software is the compiler. In UNIX, the compiler is really a collection of programs that compile sources written in the C language. In this chapter, we'll consider the following topics:

- The way the C language has evolved since its introduction and some of the problems that this evolution has caused.
- C++, an evolution of C.
- The way the compiler is organized.
- How to use the individual parts of the compiler separately, in particular the assembler and the linker.

We'll defer how the assembler and the linker work until the next chapter—to understand them, we first need to look at object files in more detail.

There are, of course, lots of other languages besides C, but on a UNIX platform C is the most important. Even if you use another language, some of the information in this chapter will be of use to you: many other languages output C source code.

The C language

The C language has evolved a lot since its appearance in the early 70's. It started life as a Real Man's language, cryptic, small, tolerant of obscenities almost to the point of encouraging them, but now it has passed through countless committees and has put on weight and become somewhat sedate, pedantic and much less tolerant. Along with this, of course, it has developed a whole lot of idiosyncracies that plague the life of the portable software writer. First, let's take a look at the flavours that are commonly available.

Kernighan and Ritchie

Kernighan and Ritchie or *K&R* is the name given to the dialect of C described in the first edition of *The C programming language* by Brian Kernighan and Dennie Ritchie. This was the first book to describe the C language, and has become something of a bible. In 1988, a second edition appeared, which describes an early version of ANSI C, *not* K&R C.

The K&R dialect is now completely obsolete, though many older versions of UNIX C resemble it. Compared to ANSI C (also called Standard C), it lacks a number of features, and has a few incompatibilities. In particular, strings were always allocated separately for each instance, and so could be modified if desired. For example, you could encounter code like this:

```
complain (msg)
char *msg;
{
  char *message = "Nothing to complain about\n";
  if (msg)                    /* parameter supplied? */
    strcpy (message, msg);    /* yes, save in message */
  puts (message);             /* say what we have to say */
  }
```

When the parameter `msg` is non-NULL, it is copied into the string `message`. If you call this function with a `NULL` message, it will display the last message again. For example:

```
complain (NULL);         prints Nothing to complain about
complain ("Bad style");  prints Bad style
complain (NULL);         prints Bad style
```

This may fail with modern C compilers: The ANSI Standard says that string constants are not writable, but real-world compilers differ in the way they handle this situation.

UNIX C

A period of over ten years elapsed between the publication of K&R and the final adoption of the ANSI C standard. During this time, the language didn't stand still, but there was no effective standards document beyond K&R. The resultant evolution in the UNIX environment is based on the Portable C Compiler first described in the paper *Portability of C Programs and the UNIX System* published by S. C. Johnson and Dennis Ritchie in 1978, and is frequently referred to as "UNIX C". This is not a standard, or even a series of standards—it's better to consider it a set of marginally compatible extensions to K&R C. You can find more information in *The evolution of C—Past and Future* by L. Rosler, but you can't rely on the degree to which your particular compiler (or the one for which your software package was written) agrees with that description. From a present-day standpoint, it's enough to know that these extensions exist, and otherwise treat the compilers like K&R. In case of doubt, the documentation that comes with the compiler is about the only even remotely reliable help. Here's a brief summary of the sort of things that had changed by the time *The evolution of C—Past and Future* appeared:

- Optional function prototyping similar to that of ANSI C was introduced. One difference exists: if a function accepts a variable number of parameters, UNIX C uses the form

  ```
  int printf (char *format, );
  ```

  whereas ANSI C uses the form

  ```
  int printf (char *format, ...);
  ```

-
- The `enum` type specifi es a way to defi ne classes of constants. For example, traditionally I could write:

  ```
  #define RED 0
  #define GREEN 1
  #define BLUE 2

  int colour;
  int x;
  colour = BLUE;
  x = RED;
  ```

  With *enum*s, I can write

  ```
  enum colours {red, green, blue};
  enum texture {rough, smooth, slimy};

  enum colours colour;
  enum texture x;
  colour = blue;
  x = red;
  ```

  This syntax is intended to make error checking easier. As you can see in the second example, there seems to be something wrong with the assignment to `x`, which was not evident in the K&R example. The compiler can see it too, and should complain, although many modern compilers compile the second program without any comment. In addition, the symbols are visible to the compiler. This means that the debugger can use them as well: preprocessor macros never make it to the code generation pass of the compiler, so the debugger doesn't know about them. The keyword `const` was added to specify that a variable may not be changed in the course of program execution.
- The preprocessor directive `#elif` was added.
- The preprocessor pseudofunction `defined (`*`identifier`*`)` was added.
- The data type `void` was added.

ANSI C

In 1989, the C programming language was finally standardized by the American National Standards Institute (ANSI) as standard X3.159-1989. In the following year it was adopted by the International Standards organization (ISO) as standard ISO/IEC 9899:1990. There are minor textual differences in the two standards, but the language defined is the same in each. The existence of two standards is good for a certain amount of confusion: some people call it ANSI C, some call it Standard C, and I suppose you could call it ISO C, though I haven't heard that name. I call it ANSI C because the name is more specific: the word "Standard" doesn't make it obvious which standard is implied.

The following discussion is intended to show the differences between ANSI C and older versions. It's not intended to teach you ANSI C—see *Practical C Programming*, by Steve Oualline, and the *POSIX Programmer's Guide* by Donald A. Lewine for that information.

ANSI C introduced a large number of changes, many of them designed to help the compiler detect program errors. You can find a reasonably complete list in Appendix C of K&R. Here are the most important from the standpoint of porting software:

- A number of changes have been made in the preprocessor. We'll look at these on page 342.
- The keywords `void`, `signed` and `const` were adopted from the Portable C compiler.
- The keyword `volatile` was added to tell an optimizer not to assume that the value of the variable will stay the same from one reference to another. Variables normally stay unchanged unless you execute an explicit assignment statement or call a function, and most optimizers rely on this behaviour. This assumption may not hold true if a signal interrupts the normal course of execution, or if you are sharing the memory with another process. If the value of a variable might change without your influence, you should declare the variable `volatile` so that the optimizer can handle it correctly. We saw an example of this kind of problem in Chapter 13, *Signals*, page 200.
- You can state the type of numeric constants explicitly: for example, you can write a long constant 0 as `0L`, and a double 0 would be `0D`.
- Implicit string literal concatenation is allowed—the following two lines are completely equivalent:

  ```
  "first string"  "second string"
  "first stringsecond string"
  ```

  K&R C allows only the second form.
- `void` pointers are allowed. Previous versions of C allowed the type `void`, but not pointers to objects of that type. You use a `void` pointer to indicate the the object you are pointing to is of indeterminate type. Before you can use the data, you need to cast it to a specific data type.
- In strict ANSI C, you must declare or define functions before you call them. You use a function *declaration* to tell the compiler that the function exists, what parameters it takes,

and what value (if any) it returns. A function *definition* is the code for the function, and includes the declaration.

Strict ANSI C function definitions and declarations include *function protyping*, which specifies the nature of each parameter, though most implementations allow old-style definitions. Consider the following function definition in K&R C:

```
foobar (a, b, c, d)
char *c;
struct baz *a;
{
body
}
```

This definition does not specify the return type of the function; it may or may not return `int`. The types of two of the parameters are specified, the others default to `int`. The parameter type specifiers are not in the order of the declaration. In ANSI C, this would become:

```
void foobar (struct baz *a, int b, char *c, int d)
{
body
}
```

This definition states all types explicitly, so we can see that `foobar` does not, in fact, return any value.

- The same syntax can also be used to declare the function, though you can also abbreviate it to:

```
void foobar (struct baz *, int, char, int);
```

This helps catch one of the most insidious program bugs: consider the following code, which is perfectly legal K&R:

```
extern foobar ();/* define foobar without parameters */
int a, b;               /* two integers */
struct baz *c;          /* and a struct baz */

foobar (a, b, c);/* call foobar (int, int, struct baz *) */
```

In this example, I have supplied the parameters to `foobar` in the wrong sequence: the `struct baz` pointer is the first parameter, not the third. In all likelihood, `foobar` will try to modify the `struct baz`, and will use the value of `a`—possibly a small integer—to do this. If I call `foobar` without parameters, the compiler won't notice, but by the time I get my almost inevitable segmentation violation, foobar will probably have overwritten the stack and removed all evidence of how the problem occurred.

Differences in the ANSI C preprocessor

At first sight, the C preprocessor doesn't seem to have changed between K&R C and ANSI C. This is intentional: for the casual user, everything is the same. When you scratch the surface, however, you discover a number of differences. The following list reflects the logical sequence in which the preprocessor processes its input.

- A method called *trigraphs* represents characters not found in the native character set of some European countries. The following character sequences are considered identical:

  Table 20−1: ANSI C trigraphs

character	trigraph
#	??=
[	??(
\	??/
]	??)
ˆ	??'
{	??<
|	??!
}	??>
˜	??-

  To show what this means, let's look at a possibly barely recognizable program:

  ```
  ??=include <unistd.h>
  main ()
    ??<
    printf ("Hello, world??/n");
    ??>
  ```

  Not surprisingly, most programmers hate the things. To quote the *gcc* manual: "*You don't want to know about this brain-damage*". Many C compilers, including the GNU C compiler, give you the opportunity to turn off support for trigraphs, since they can bite you when you're not expecting them.

- Any line may end with `\`, indicating that it should be *spliced*—in other words, the preprocessor removes the `\` character and the following newline character and joins the line to the following line. K&R C performed line splicing only during the definition of preprocessor macros. This can be dangerous: trailing blanks can nullify the meaning of the \ character, and it's easy to oversee one when deleting lines that follow it.

- Unlike UNIX C, formal macro parameters in strings are not replaced by the actual parameters. In order to be able to create a string that includes an actual parameter, the operator # was introduced. A formal parameter preceded by a # is replaced by the actual parameter surrounded by string quotes. Thus

```
#define foo(x)  open (#x)
foo (/usr/lib/libc.a);
```

will be replaced by

```
open ("/usr/lib/libc.a");
```

In many traditional versions of C, you could have got the same effect from:

```
#define foo(x)  open ("x")
foo (/usr/lib/libc.a);
```

- In K&R C, problems frequently arose concatenating two parameters. Since both parameter names are valid identifiers, you can't just write one after the other, because that would create a new valid identifer, and nothing would be substituted. For example, consider the X11 macro `Concat`, which joins two names together to create a complete path name from a directory and a file name:

  ```
  Concat(dir, file);
  ```

  I obviously can't just write

  ```
  #define Concat(dir, file) dirfile
  ```

  because that will always just give me the text `dirfile`, which isn't much use. The solution that the X Consortium used for K&R C was:

  ```
  #define Concat(dir,file)dir/**/file
  ```

  This relies on the fact that most C compilers derived from the portable C compiler simply remove comments and replace them by nothing. This works most of the time, but there is no basis for it in the standard, and some compilers replace the sequence `/**/` with a blank, which breaks the mechanism. ANSI C introduced the operator *##* to address this problem. *##* removes itself and any white space (blanks or tab characters) to either side. For ANSI C, *Imake.tmpl* defines `Concat` as

  ```
  #define Concat(dir,file)dir##file
  ```

- The *#include* directive now allows the use of preprocessor directives as an argument. *imake* uses this to *#include* the *<vendor>.cf* file.

- Conditional compilation now includes the `#elif` directive, which significantly simplifies nested conditional compilation. In addition, a number of logical operators are available: `||` and *&&* have the same meaning as in C, and the operator `defined` checks whether its operand is defined. This allows code like:

  ```
  #if defined BSD || defined SVR4 || defined ULTRIX
  foo
  #elif defined SVR3
  bar
  #endif
  ```

  If you want, you can surround the operand of `defined` with parentheses, but you don't need to.

- The use of the preprocessor directive `#line`, which had existed in previous versions of C, was formalized. `#line` supports preprocessors that output C code—see page 88 for an example. `#line` tells the compiler to reset the internal line number and file name used for error reporting purposes to the specified values. For example if the file *bar.c* contains just

  ```
  #line 264 "foo.c"
  slipup!
  ```

  the compiler would report the error like this:

  ```
  $ gcc -O bar.c -o bar
  foo.c:264: parse error before ‘!’
  gnumake: *** [bar] Error 1
  ```

  Although the error was really detected on line 2 of *bar.c*, the compiler reports the error as if it had occurred on line 264 of *foo.c*.

- The line *slipup!* suggests that it is there to draw attention to an error. This is a fairly common technique, though it’s obviously just a kludge, especially as the error message requires you to look at the source to figure out what the problem is. ANSI C introduced another directive to do the Right Thing. Instead of `slipup!`, I can enter:

  ```
  #error Have not finished writing this thing yet
  ```

  This produces (from *gcc*)

  ```
  $ make bar
  gcc -O bar.c -o bar
  foo.c:270: #error Have not finished writing this thing yet
  gnumake: *** [bar] Error 1
  ```

  I couldn’t write **`Haven’t`**, because that causes *gcc* to look for a matching apostrophe (`’`). Since there isn’t one, it would die with a less obvious message, whether or not an error really occurred.

- To quote the Standard:

  A preprocessor line of the form **`# pragma`** *token-sequence*$_{opt}$ *causes the processor to perform an implementation-dependent action. An unrecognized pragma is ignored.*

This is not a Good Thing. Implementation-dependent actions are the enemy of portable software, and about the only redeeming fact is that the compiler ignores an unrecognized pragma. Since almost nobody uses this feature, you can hope that your compiler will, indeed, ignore any pragmas it finds.

Assertions

Assertions provide an alternative form of preprocessor conditional expression. They are specified in the form

```
#assert question (answer)
```

In the terminology of the documentation, this *asserts* (states) that the answer to *question* is *answer*. You can test it with the construct:

```
#if #question(answer)
...
#endif
```

The code between `#if` and `#endif` will be compiled if the answer to `question` is `answer`. An alternative way to use this facility is in combination with the compiler directive `-Aquestion(answer)`. This method is intended for internal use by the compiler: typically, it tells the compiler the software and platform on which it is running. For example, compiling *bar.c* on UNIXWare 1.0 with *gcc* and the `-v` flag reveals:

```
/usr/local/lib/gcc-lib/i386-univel-sysv4.2/2.4.5/cpp \
-lang-c -v -undef -D__GNUC__=2 -Di386 -Dunix -D__svr4__ \
-D__i386__ -D__unix__ -D__svr4__ -D__i386 -D__unix \
-D__svr4__ -Asystem(unix) -Acpu(i386)  -Amachine(i386) \
bar.c /usr/tmp/cca000Nl.i
```

The `-A` flags passed by *gcc* to the preprocessor specify that this is a `unix` system and that the cpu and machine are both `i386`. It would be nice if this information stated that the operating system was `svr4`, but unfortunately this is not the default for System V.4. *gcc* has also retrofitted it to System V.3, where the assertion is `-Asystem(svr3)`, which makes more sense, and to BSD systems, where the assertion is `-Asystem(bsd)`.

C++

C++ is an object-oriented evolution of C that started in the early 80's, long before the ANSI C standard evolved. It is almost completely upwardly compatible with ANSI C, to a large extent because ANSI C borrowed heavily from C++, so we don't run much danger by considering it the next evolutionary step beyond ANSI C.

The last thing I want to do here is explain the differences between ANSI C and C++: *The Annotated C++ Reference Manual*, by Margaret A. Ellis and Bjarne Stroustrup, spends nearly 450 very carefully written pages defining the language and drawing attention to its peculiarities. From our point of view, there is not too much to say about C++.

One of the more popular C++ translators is AT&T's *cfront*, which, as the name suggests, is a front-end preprocessor that generates C program code as its output. Although this does not make the generated code any worse, it does make debugging much more difficult.

Since C++ is almost completely upwards compatible from ANSI C, a C++ compiler can usually compile ANSI C. This assumes well-formed ANSI C programs: most ANSI C compilers accept a number of anachronisms either with or without warnings—for example, K&R-style function definitions. The same anachronisms are no longer part of the C++ language, and cause the compilation to fail.

C++ is so much bigger than C that it is not practicable to even think about converting a C++ program to C. Unless there are some really pressing reasons, it's a whole lot easier to get hold of the current version of the GNU C compiler, which can compile both C and C++ (and Objective C, if you're interested).

C and C++ have different function linking conventions. Since every C++ program calls C library functions, there is potential for errors if you use the wrong calling convention. We looked at this aspect in Chapter 17, *Header files*, on page 286.

Other C dialects

Before the advent of ANSI C, the language was ported to a number of non-UNIX architectures. Some of these added incompatible extensions. Many added incompatible library calls. One area is of particular interest: the Microsoft C compiler, which was originally written for MS-DOS. It was subsequently adapted to run under XENIX and SCO UNIX System V. Since our emphasis is on UNIX and UNIX-like systems, we'll talk about the XENIX compiler, though the considerations also apply to SCO UNIX and MS-DOS.

The most obvious difference between the XENIX C compiler and most UNIX compilers is in the flags, which are described in Appendix B, *Compiler flags*, but a couple of architectural limitations have caused incompatibilities in the language. We'll look at them in the following section.

Intel 8086 memory models

The original MS-DOS compiler ran on the Intel 8086 architecture. This architecture has 1 MB of real memory, but the addresses are only 16 bits long. In order to address memory, each machine instruction implicitly adds the contents one of four *segment registers* to the address, so at any one time the machine can address a total of 256 kB of memory. In order to address more memory, the C implementation defi nes a 32 bit pointer type, the so-called *far address*, in software. Accessing memory via a far pointer requires reloading a segment register before the access, and is thus signifi cantly slower than access via a 16-bit *near address*. This has a number of consequences:

Near addresses are simply offsets within a segment: if the program expects it to point to a different segment, it will access the wrong data.

Far pointers are 32 bits wide, containing the contents of the segment register in one half and the offset within the segment in the other half. The segment register contains bits 4 through 19 of a 20-bit address, and the offset contains bits 0 through 15. To create an absolute address from a far pointer, the hardware performs effectively

```
struct fp
  {
  short segment_reg;          /* 16 bits, bits 4 through 19 of address */
  short offset;               /* 16 bits, bits 0 through 15 of address */
  }
long abs_address = (fp.segment_reg << 4) + fp.offset;
```

As a result, many possible far pointer contents that could resolve to the same address. This complicates pointer comparison signifi cantly. Some implementations solved this problem by declaring *huge* pointers, which are normalized 20-bit addresses in 32-bit words.

Along with three pointer types, MS-DOS C uses a number of different executable formats. Each of them has default pointer sizes associated with them. You choose your model by supplying the appropriate flag to the compiler, and you can override the default pointer sizes with the explicit use of the keywords `near`, `far` or (where available) `huge`:

- The *tiny model* occupies a single segment and thus can always use near addresses. Apart from the obvious compactness of the code, this model has the advantage that it can be converted to a *.COM* file.
- The *small model* occupies a single data segment and a single code segment. Here, too, you can always use near pointers, but you need to be sure you're pointing into the correct segment.
- The *medium model* (sometimes called *middle model*) has multiple code segments and a single data segment. As a result, code pointers are far and data pointers are near.
- The *compact model* is the inverse of the medium model. Here, code is restricted to one segment, and data can have multiple segemnts. Static data is restricted to a single segment. As a result, code pointers are near and data pointers are far.
- The *large model* can have multiple code and multiple data segments. Static data is restricted to a single segment. All pointers are far.
- The *huge model* is like the large model except that it can have multiple static data segments. The name is unfortunate, since it suggests some connection with huge pointers. In fact, the huge model uses far pointers.

What does this mean to you? If you're porting from MS-DOS to UNIX, you may run into these keywords `near`, `far` and `huge`. This isn't a big deal: you just need to remove them, or better still, define them as an empty string. You may also find a lot of pointer checking code, which will probably get quite confused in a UNIX environment. If you do find this kind of code, the best thing to do is to ifdef it out (`#ifndef unix`).

If you're converting from UNIX to MS-DOS, things can be a lot more complicated. You'll be better off using a 32-bit compiler, which doesn't need this kind of kludge. Otherwise you may have to spend a considerable amount of time figuring out the memory architecture most suitable for your package.

Other differences in MS-DOS

MS-DOS compilers grew up in a very different environment from UNIX. As a result, a number of detail differences exist. None of them are very serious, but it's good to be forewarned:

- They do not adhere to the traditional UNIX organization of preprocessor, compiler, assembler and loader.
- They don't use the assembler directly, though they can usually output assembler code for use outside the compilation environment.

- The assembler code output by MS-DOS compilers is in the standard Intel mnemonics, which are not compatible with UNIX assemblers.
- Many MS-DOS compilers combine the preprocessor and the main compiler pass, which makes for faster compilation and less disk I/O.
- Many rely on the Microsoft linker, which was not originally written for C, and which has significant limitations.
- Many MS-DOS compilers still run in real mode, which limits them to 640K code and data. This is a severe limitation, and it is not uncommon to have to modify programs in order to prevent the compiler from dying of amnesia. This leads to a different approach with header files, in particular: in UNIX, it's common to declare everything just in case, whereas in MS-DOS it may be a better idea to not declare anything unless absolutely necessary.

Compiler organization

The traditional UNIX compiler is derived from the Portable C Compiler and divides the compilation into four steps, traditionally called *phases* or *passes*, controlled by the compiler control program *cc*. Most more modern compilers also adhere to this structure:

1. The *preprocessor*, called *cpp*, reads in the the source files and handles the preprocessor directives (those starting with #) and performs macro substitution.
2. The compiler itself, usually called *cc1*, reads in the preprocessor output and compiles to assembler source code. In SunOS, this pass is called *ccom*.
3. The assembler *as* reads in this output and assembles it, producing an object file.
4. The loader takes the object file or files and links them together to form an executable. To do so, it also loads a low-level initialization file, normally called *crt0.o*, and searches a number or libraries.

cc usually performs these passes invisibly. The intermediate outputs are stored in temporary files or pipes from one pass to the next. It is possible, however, to call the passes directly or to tell *cc* which pass to execute—we'll look at how to do that in the next section. By convention, a number of suffixes are used to describe the intermediate files. For example, the GNU

C compiler recognizes the following suffi xes for a program *foo*:

Table 20−2: C compiler intermediate files

fi le	contents	created by compiler?
foo.c	unpreprocessed C source code	
foo.cc	unpreprocessed C++ source code	
foo.cxx	unpreprocessed C++ source code	
foo.C	unpreprocessed C++ source code	
foo.i	preprocessed C source code	yes
foo.ii	preprocessed C++ source code	yes
foo.m	Objective C source code	
foo.h	C header fi le	
foo.s	assembler source code	yes
foo.S	assembler code requiring preprocessing	
foo.o	object fi le	yes

Here's what you need to do to go through the compilation of *foo.c* to the executable *foo*, one pass at a time:

```
$ gcc -E foo.c -o foo.i      preprocess
$ gcc -S foo.i               compile
$ gcc -c foo.s               assemble
$ gcc foo.o -o foo           link
```

There are slight variations in the form of the commands: if you don't tell the preprocessor where to put the output fi le, *gcc* writes it to *stdout*. Other preprocessors may put a special suffi x on the base fi le name, or if you specify the `-o` flag, the compiler might put it in the fi le you specify. If you don't tell the linker where to put the output fi le, it writes to *a.out*.

Compiling an object fi le from an assembler fi le is the same as compiling from a source fi le or a preprocessed fi le—*gcc* decides what to do based on the suffi x of the input fi le.

You can also run any combination of contiguous passes like this:

```
$ gcc -S foo.c               preprocess and compile
$ gcc -c foo.c               preprocess, compile and assemble
$ gcc -o foo foo.c           preprocess, compile, assemble, link
$ gcc -c foo.i               compile and assemble
$ gcc -o foo foo.i           compile, assemble, link
$ gcc -o foo foo.s           assemble and link
```

The location of the C compiler is, unfortunately, anything but standardized. The control program *cc* is normally in */usr/bin*, or occasionally in */bin*, but the other components might be stored in any of the following: */usr/lib*, */usr/ccs/lib* (System V.4), */usr/lib/cmplrs/cc* (MIPS) or */usr/local/lib/gcc-lib* (*gcc* on most systems).

Other compiler organizations

Some modern compilers have additional passes. Some optimizers fit between the compiler and the assembler: they take the output of the compiler and output optimized code to the assembler. An extreme example is the MIPS compiler, which has a total of 8 passes: The preprocessor *cpp*, the front end *cc1*, the *ucode*[*] linker *uld*, the procedure merge pass *umerge*, the global optimizer *uopt*, the code generator *ugen*, the assembler *as1*, and the linker *ld*. Despite this apparent complexity, you can consider this compiler as if it had only the traditional four passes: the five passes from the front end up to the code generator perform the same function as the traditional *cc1*.

The C preprocessor

You can use the preprocessor *cpp* for other purposes than preprocessing C source code: it is a reasonably good macro processor, and it has the advantage that its functionality is available on every system with a C compiler, though in some cases it is available only via the C compiler. It is one of the mainstays of *imake*, and occasionally packages use it for other purposes as well.

There are two ways to invoke *cpp*: you can invoke it with *cc* and the `-E` flag, or you can start it directly. If at all possible, you should start it via *cc* rather than running it directly. On some systems you can't rely on *cc* to pass the correct flags to *cpp*. You also can't rely on all versions of *cpp* to use the same flags—you can't even rely on them to be documented. You can find a list comparing the more common preprocessor flags in Appendix B, *Compiler flags*, page .

Which compiler to use

Most systems still supply a C compiler, and normally this is the one you would use. In some cases, bugs in the native system compiler, compatibility problems, or just the fact that you don't have the normal compiler may lead to your using a different compiler. This situation is becoming more common as software manufacturers unbundle their compilers.

Using a different compiler is not necessarily a Bad Thing, and can frequently be an improvement. In particular, *gcc*, the GNU C compiler from the Free Software Foundation, is very popular—it's the standard C compiler for a number of systems, including OSF/1, 4.4BSD, and Linux. It can do just about everything except run in minimal memory, and it has the advantage of being a well-used compiler: chances are that somebody has compiled your package with *gcc* before, so you are less likely to run into trouble with *gcc* than with the native compiler of a less-known system. In addition, *gcc* is capable of highly optimized code, in many cases significantly better than the code created by the native compiler.

Compilers are becoming more standardized, and so are the bugs you are liable to run into. If you have the choice between compiling for K&R or ANSI, choose ANSI: the K&R flags may

[*] *ucode* is a kind of intermediate code used by the compiler. It is visible to the user, and you have the option of building and using ucode libraries.

use “features” that were not universally implemented, whereas the ANSI versions tend to pay more attention to the standard. If you do run into a bug, chances are someone has seen it before and has taken steps to work around it. In addition, compiling for ANSI usually means that the prototypes are declared in ANSI fashion, which increases the chance of subtle type conflicts being caught.

Some things that neither you nor the *Makefile* may expect are:

- *gcc* compiles both K&R (`-traditional`) and ANSI dialects. However, even some software supplied by the Free Software Foundation breaks when compiled with *gcc* unless the `-traditional` flag is used.
- Many compilers do not compile correctly when both optimization and debugging information are specified (`-O` and `-g` flags), though most of them recognize the fact and turn off one of the flags. Even if the compiler ostensibly supports both flags together, bugs may prevent it from working well. For example, *gcc* version 2.3.3 generated invalid assembler output for System V.4 C++ programs if both flags were specified. Even when compilers do create debugging information from an optimizing compilation, the results can be confusing due to the action of the optimizer:
    - The optimizer may remove variables. As a result, you may not be able to set or display their values.
    - The optimizer may rearrange program flow. This means that single-stepping might not do what you expect, and you may not be able to set breakpoints on certain lines because the code there has been eliminated.
    - Some optimizers remove stack frames,* which makes for faster code, particularly for small functions. *gcc* will do this with the `-O3` option.

  Stack frame removal in particular makes debugging almost impossible. These aren’t bugs, they’re features. If they cause problems for you, you will need to recompile without optimization.
- Some compilers limit the length of identifiers. This can cause the compiler to treat two different identifiers as the same thing. The best thing to do if you run into this problem is to change the compiler: modern compilers don’t have such limits, and a compiler that does is liable to have more tricks in store for you.
- With a System V compiler, you might find:

  ```
  $ cc -c frotzel.c -o frotzel.o
  cc: Error: -o would overwrite frotzel.o
  ```

  System V compilers use the flag `-o` only to specify the name of the final executable, which must not coincide with the name of the object file. In many *Makefiles* from the BSD world, on the other hand, this is the standard default rule for compiling from *.c* to *.o*.

* See Chapter 21, *Object files and friends*, page 377, for further information on stack frames.

- All C compilers expect at least some of their flags in a particular sequence. The documentation is frequently hazy about which operands are sequence-sensitive, or what interactions there are between specific operands.

The last problem bears some more discussion. A well-documented example is that the linker searchs library specifications (the `-l` option) in the sequence in which they are specified on the compiler invocation line—we'll investigate that in more detail in Chapter 21, *Object files and friends*, page 373. Here's an example of another operand sequence problem:

```
$ cc foo.c -I../myheaders
```

If *foo.c* refers to a file *bar.h* in the directory *../myheaders*, some compilers won't find the header because they don't interpret the `-I` directive until after they try to compile *foo.c*. The man page for System V.4 *cc* does state that the compiler searches directories specified with `-I` in the order in which they occur, but it does not relate this to the sequence of operands and file names.