C for Yourself

Programming in C under UNIX

Graham Wheeler
Riël Smit

Department of Computer Science
University of Cape Town

This documentation was prepared with LaTeX and reproduced by the Printing Department, University of Cape Town from camera-ready copy supplied by the authors.

Preface

C is a small language, and yet it is not simple to master. This is because the source of the power of C, namely the freedom it allows the programmer to do what (s)he wants, is also the source of its weakness: it may fail catastrophically.

Programming in a language like Pascal or Modula-2 is like driving around a city in a sophisticated car, with excellent brakes and other safety features. Programming in C is like flying a microlight. Your controls are few and simple, yet you can soar above the restrictions of roads and traffic signs, and get to your destination much more directly. However, when you make an error, the results are catastrophic. Obviously, the better you can control the microlight, the greater its advantages. This control comes not just from a sound training, but also from experience. It is the aim of this book to provide the former; it is up to the reader to obtain the latter.

The book is written for programmers who are familiar with some other structured language (such as Pascal or more recent dialects of Basic), and who are moving to the UNIX system and need to learn C. It covers the entire C language, as well as the most common library functions which give C its power. In addition we have included appendices on the main UNIX editor vi, as well as many of the UNIX programmer's support tools for debugging and program maintenance. These tools greatly simplify the task of developing programs in C.

Recently, in an attempt at developing a standard for the language, the C language has been undergoing changes. Many of these changes have not yet filtered down into the commercial UNIX world yet, so we have concentrated on the most common version of C, so-called Kernighan and Ritchie C. However, we have mentioned throughout the text areas where changes are occurring, and have included an appendix listing the main enhancements that have been made. These enhancements are very welcome, and if your compiler supports them, you would do well to use them, in particular function prototypes.

We have tried to make the book as comprehensive as possible. Thus, the book is not only an introduction for programmers learning C, but should remain a useful reference, even for experienced C programmers. Alas, similar to programs, we know that there are errors and omissions in this book. We would appreciate being notified of any that you find, regardless of how trivial you may think they are.

In conclusion, may you come to "C for Yourself" what can, what should, and what should not be done with C.

Graham Wheeler
Riël Smit
Cape Town, February 1990.

Preface to 2nd Printing

Apart from correcting all the errors we were aware of, we added information on initialisation of arrays, input/output of structures, placing structures at absolute addresses, bitfields, and startup code. Furthermore, two articles dealing with portable code and C style have been included as appendices. These articles have been included as they were received electronically (with only minor changes to allow them to be formatted with the rest of the book). Their typographical style is therefore not necessarily the same as that of the rest of the book. Neither have we implemented all the C style recommendations. They are only recommendations after all. `:-)`

GW & RS
Cape Town, February 1991.

Contents

Chapter 1

Introduction

1.1 A Short History of C and UNIX

In the 1960's, as the use of computers changed from batch processing systems to the new concept of time-sharing, operating system design was a major computing issue. In England, the Universities of London and Cambridge embarked on the Combined Programming Language (CPL) project, an attempt to provide a single language for all applications, including what were previously thought to be assembly language applications.

CPL was not entirely successful, and never came into general use, but it did inspire the language BCPL (Basic CPL). BCPL is a system's programming language with the main distinguishing feature that it is typeless - the only data objects allowed are machine words; it is up to the programmer to implement types such as characters, strings, real numbers, and so on.

In America, much work was being done on the Multics (MULTiplexed Information and Computing Service) project, a joint Massachusetts Institute of Technology, Bell Laboratories and General Electric Corporation time-sharing project started in 1965 as part of MIT's "Project MAC." After three years, no usable system had yet been produced, and Bell Laboratories withdrew from the project. Ken Thompson and Dennis Ritchie were two of the last Bell Labs employees still working on Multics, and were unhappy with the idea of losing the limited time-sharing capability Multics had thus far produced. They began designing their own file system on paper. Thompson found an unused PDP-7 to play with, and their file system was soon bootstrapped onto it. They were encouraged in their task by the fact that CPU time on the mainframe for Thompson's elaborate "Space Travel" game cost about $75 a game, while the PDP-7 was theirs to play with and had superior graphics!

Some essential utilities for file handling, including printing and editing, were added, as well as a way to run system processes. In 1970, Brian Kernighan named this rudimentary system "UNIX" – a pun on Multics.

The UNIX system was moved onto a PDP-11 for developing a text processing system for Bell, and the First Edition documentation written by Ritchie and Thompson in late 1971, the Second Edition appearing in June 1972.

B, a direct descendant of BCPL, was used by Thompson for the Second Edition assembler. He extended B into a language called NB, in the hope of using it to rewrite UNIX. However, this proved unsuccessful. Ritchie, attempting to improve the performance of NB, came up with a similar language, which he called C. In 1973 structures were added to C, adding considerable power, and UNIX was rewritten in C.

Due to a government agreement, Bell Labs could not market UNIX, but provided it free to universities, which resulted in a large growth of software, particularly at the University of California at Berkeley.

In 1978 Kernighan and Ritchie published *The C Programming Language* (Prentice-Hall), still considered the definitive reference on C. A summary from this book is reprinted as "The C Programming Language - Reference Manual" by Dennis Ritchie (see "UNIX Programmer's Manual volume 2", Bell Laboratories).

Although there is an American National Standards Institute (ANSI) proposed standard, C dialects abound. However, most implementations conform closely to the 1978 version (known as Kernighan & Ritchie, or K&R C) and, more increasingly, the ANSI standard recommendations, although these are less commonly found on UNIX systems. Steve Johnson's Portable C Compiler (PCC), the basis of many UNIX C compilers, is also often used as a standard reference.

Hand-in-hand with the development of UNIX and C was the development of their underlying philosophy, dubbed the "Software Tools" philosophy. This philosophy will be discussed in more detail later.

1.2 The UNIX cc Compiler

The cc compiler is the main compiler of the UNIX system, and comprises several parts. These parts are often transparent to the user, since the cc command calls them as necessary. Nonetheless, it is useful to know a little bit about the compiler and its constituent phases.

First, there is the *preprocessor*, cpp. This is a simple macroprocessor, which has a small set of commands allowing macros to be defined, sections of program to be conditionally compiled, text from other files to be included, and so on. cpp also strips comments from files. The output from cpp is passed as a text stream to the first pass of the compiler proper. The preprocessor is described in more detail in the next chapter.

The compiler conceptually operates in two passes, although on most systems these actually occur together. The first pass takes the output from cpp and converts it into an intermediate file. This file typically contains assembly language statements for the higher level program code (such as loops), and a symbolic representation of the lower level expressions in the program. The second pass copies the assembly language statements to a further intermediate file, and generates assembly language for the expressions. There is also an optional optimisation phase, c2, after the second pass.

The compiler output is given to the *assembler*, as. This is the resident UNIX assembler and varies from system to system. The assembler converts the assembly language file into machine code, and saves the results in a file using the UNIX common object file format (or COFF). This format includes various headers, symbol table and relocation information, and three sections - one for the machine code (called **.text**), one for the initialised data in the program (called **.data**), and one for uninitialised data (called **.bss**).

The assembler output file is then passed to the *linker*, ld. This is the standard UNIX linker. It adds any necessary library functions to the file, and resolves references across files if the program is split over several files.

The final executable file is always put in a file called **a.out**, unless you specify otherwise. The name of the C source file should always end with a **.c** suffix. Two of the compilation phases create intermediate files with the same name as the source file, but with different suffixes, namely:

.s	the output from the second pass of the compiler (assembly language)
.o	the output from the assembler (machine code)

These intermediate files are usually destroyed, unless you specify otherwise. For details on the compiler options available, consult Appendix A.

Chapter 2

The C Programming Language

A C program consists of *preprocessor commands*, *comments*, some *top-level declarations*, and one or more *functions*, possibly spread over more than one file. One and only one of the functions must be called `main`; this is always the first function to be executed. Functions may refer to (or call) one another, with the exception of `main`, which cannot be called by any other function. The order of the functions in the file is incidental; they are typically grouped together according to their purpose. The next section describes the preprocessor and all but the first paragraph may be skipped on the first reading of this book.

2.1 The Preprocessor

The C preprocessor is a simple macroprocessor, controlled by special command lines within the source program. Each command line begins with a hash (`#`); in most compilers, this must be the first character on the line. The common commands are:

`#define`	define a macro
`#undef`	remove a macro definition
`#include`	include another file
`#if, #ifdef,`	conditional ...
`#ifndef, #else,`	...test...
`#endif`	... commands

The preprocessor executes all such commands it finds in the source file, removing them as it does so. Preprocessor commands can be split over more than one line by ending each line except the last with a backslash (\).

2.1.1 Macro Definitions

Macro definitions provide a way of naming useful constants, as well as improving readability and abbreviating common short pieces of text. A macro definition has the form

`#define` *macro_name* [(*arguments*)] [*macro_body*]

where [] indicate optional parts.

Once a macro has been defined, and until the end of the file containing the definition or an `#undef` command for the same macro, *all occurrences of the macro name in the file will be replaced by the macro body.* If arguments are used, these will be substituted into the macro body.

Some examples will help. Firstly, naming useful constants:

```
#define TRUE        1
#define FALSE       0
#define PI          3.1415927
#define BUFFERSIZE  80
#define STAR        '*'
#define ERROR       -1
```

These are reasonably obvious examples. As far as the first two are concerned, C provides no reserved words for the Boolean values true and false, but treats zero as being false and everything else as being true, with one being the canonical true value. Thus, by defining these macros, we can use `TRUE` and `FALSE` in our program instead of the more obscure 1 and 0. The `BUFFERSIZE` example illustrates a technique for improving readability (as with `TRUE` and `FALSE`), as well as ease of modification. This constant would presumably be used when declaring line buffers, for example. If the desired size of the line buffer changed, it would only be necessary to change the macro definition, instead of searching for the number 80 and changing it to the new number throughout the file, which is an error-prone method.

Examples of macros taking arguments are:

```
#define  square(x)             ((x)*(x))
#define  cube(x)               ((x)*(x)*(x))
#define  hypotenu(a,b)         (sqrt(square(a)+square(b)))
#define  distance(x1,y1,x2,y2)  (hypotenu(x1-x2,y1-y2))
```

Macros thus offer a simple alternative to functions[1]. The difference is that a function is compiled once, and every time it is referenced, a call is made to that same single function. A macro, however, is simply *a textual replacement before compilation even begins*. Thus every time the macro is used, the compiler will generate code for the whole macro. The advantage of macros is that they make programs easier to read and modify. In general, the use of complex macros is discouraged, as a function would almost always serve the purpose better. However, a macro is generally better than a trivial function.

Macros can refer to one another, as was shown above: when a macro reference occurs in a program, the macro body is substituted, and the preprocessor carries on from the *beginning* of the substituted text. However, a macro cannot refer to another macro which has not yet been defined.

Warning: You should use parentheses in macros wherever they could be ambiguous. This is surprisingly often. Take, for example, the following macro:

```
#define SQUARE(x)     x*x
```

If this was used in the program as:

```
SQUARE(z+1)
```

this would expand to:

```
z+1*z+1
```

or `z+z+1`, which is hardly what we want. The safest definition here is the one given earlier for `square(x)`.

To remove a definition, use the `#undef` command as in

```
#undef SQUARE
```

By convention upper case letters are usually used for macros that are simply constants (e.g. PI). Macros that take arguments are often regarded as being similar to functions, and the convention then does not apply.

2.1.2 Including Text From Other Files

The command

[1]A function in C is a group of statements associated with a name, by which they can be called and executed.

```
#include <filename>
```

causes the entire content of the specified file (the *include file*) to be processed as if it had appeared in place of the command. It may be that the included file itself contains `#include` commands; the depth to which this is possible varies from system to system.

File names must be in double quotes (`""`) or angle brackets (`<>`). The quotes are used for user files, the angle brackets for UNIX system files. The difference is that files enclosed in quotes are first searched for in the same directory as the program, and then in the standard directories, while files enclosed in angle brackets are searched for in the standard directories only. Under UNIX, the standard include directory is `/usr/include`.

There is a standard input/output header file, called **stdio.h**, which must be included by any C program that performs input or output. As it is a short file, and defines a few useful constants, it is best at first to always include it. You should print a copy of your system's **stdio.h** file and examine it to see what constants it defines (for example, some define the `TRUE` and `FALSE` constants we defined above, while others do not).

2.1.3 Conditional Compilation

Conditional commands allow parts of the program to be included or excluded from compilation, depending on some condition. As the preprocessor must be able to determine the condition, only simple constant expressions are allowed, with the usual C convention that if the expression has value zero, it is considered false, otherwise true. The main use of conditional compilation is to make programs more portable, and to (conditionally) include debugging code in a program. The general form of a conditional is:

```
#if constant_expression
    text_section_one
#else
    text_section_two
#endif
```

Constant expressions will be described in detail later; for now it is enough to say that *constant_expression* is an expression that can be evaluated immediately from the information known by the compiler (or, in this case, the preprocessor). If the expression is true, *text_section_one* is included by the preprocessor and *text_section_two* is ignored; if the expression is false, it is the other way around.

Two other forms of `#if` are available, namely:

`#ifdef` *macro_name*

and

`#ifndef` *macro_name*

These test whether the macro name has been defined or is undefined, respectively. This could be of use, for example, if we are unsure whether `TRUE` has been defined. We could use:

```
#ifndef  TRUE
#define  TRUE 1
#endif
```

2.2 Comments

Comments in C are enclosed within `/*` and `*/`. Comments can continue over more than one line. You should not use comments within comments; `/*` is not recognised within a comment, so (usually) the first `*/` found indicates the end of comments. However, this is compiler dependent and not all compilers may act this way.

2.3 Functions

A function is a collection of *declarations* and *statements* enclosed in braces {}, and preceded by a *function name.* The minimal C function, which does nothing[2] (it contains no declarations or statements), is:

```
tiny()
{
}
```

The parentheses following the name are necessary; they are used to enclose the names of the function's arguments, if any. Even if there are no arguments, they serve to indicate to the compiler that the name is that of a function, and not a variable. Function names must be valid C *identifiers*.

[2]This is not quite true, because the function, as declared here, returns a value by default, but more about this later.

2.3.1 Identifiers

Identifiers in C are names for functions and variables. They may be any combination of upper and lower case letters, digits (0–9), and the underscore character (_), the only restriction being that they may not begin with a digit (this is so that the compiler can recognise them as identifiers and not numbers). In older compilers only the first eight characters are significant, thus the compiler might not distinguish between `long_name_1` and `long_name_2`. The case of the letters is significant, thus `Total` and `total` are different identifiers (it is considered bad style to have two identifiers that differ only in their case).

All identifiers are characterised by two attributes: their *type* and their *storage class*. The type represents the identifier type, for example, `int` (integer), `float` (floating point number), `char` (character), or function. The storage class determines the identifier's *scope* and *visibility*; that is, how accessible the identifier is from other parts of the program. The main storage classes are:

external: an external identifier can be referenced from anywhere in the program, even if the program is split over several files. The linker (ld) knows the details of all external identifiers and can thus resolve references across files. All functions and top-level declarations are external unless otherwise specified. Memory for external variables is allocated once only, at compile time. Another name for external identifiers is *global identifiers*.

automatic: this is the default storage class for all variables declared within blocks. Top-level declarations of automatic variables are not allowed. An automatic variable can only be accessed within the block containing its declaration. It is called automatic because memory is automatically allocated for it when the block is entered and given up when the block is exited (that is, at run time, not compile time as in external). This means that the value of an automatic variable is lost when the function is exited (they are *volatile*). The linker does not need any information about automatic variables. Another name for automatic variables is *local variables*, as they are local to the block that contains them.

static: static variables are much the same as external variables. Like externals, they have storage allocated for them once, at compile time. Thus the value of a static variable is *not* lost when the containing block is exited (it is *non-volatile*). However, a static identifier is not visible outside its containing block, is not available to the linker, and thus cannot be accessed across files. They thus provide a way of limiting the sharing of non-volatile variables. A typical use of static identifiers is for use in execution profiles – counting the number of times a function is executed. Each function can have a static variable which has one added each time the function is entered. Such a counter can also be used by a function to determine the first time it is being executed and act appropriately. For example, an output routine can open a file for output the first time it is called.

2.3.2 Simple Declarations and Statements

Before we can begin using functions, we need variables to hold the values we want to work upon. All variables in C must be declared before being used. This tells the compiler necessary details about the variable, such as the type and storage class. At this stage, we will just consider very basic declarations of integer and character variables. Such a declaration has the general form

storage_class type list_of_identifiers;

where *storage_class* is optional. The default storage class specifier is external for top level declarations and functions, and automatic for any other declarations. If *storage_class* is present, it should be one of `extern`, `auto`, `static`, or `register` (the latter will be described later). The *type* specifier is essential and in this case should be either `int` or `char`. *list_of_identifiers* must contain at least one identifier name, but may have more, in which case each identifier must be separated by commas. Like all C statements, a declaration is terminated by a semicolon. For example:

```
int x,y,result;
static char c, next_c;
```

Having declared these variables, we can use them in simple *assignment statements*, for example:

```
result = x+y;
c = '*';
```

Note the use of `=` for assignment as opposed to Pascal's `:=`, as well as single quotes to enclose *character constants* in C as opposed to double quotes, which are used for *string constants*.

In order to make our programs produce meaningful results, we need some way of entering and printing the values of these variables.

2.3.3 Basic Character and Integer Input/Output

Input and output statements, strangely enough, are not a part of the C language itself. C is a small language, and many of its features are actually provided by precompiled functions in special *system libraries.* The linker searches the libraries and extracts only those functions used by the program, adding them to the compiled program. There are libraries for I/O functions, string handling, math functions, and so on. To use the standard input/output runtime library, the standard input/output header file, **stdio.h**, must be included in the source file using:

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

This provides access to numerous functions. Two of the most powerful (although their full power will only become apparent later) are `printf` for output and `scanf` for input.

`printf` takes at least one argument – a string enclosed in quotes. This is the string which `printf` prints. However, within this string we can use `%d` to represent a place to print an integer in decimal, and `%c` to represent a place to print a character. The appropriate variables or constants must then be included as arguments to `printf`, in the correct order. For example, assuming `x` has value 3, `y` has value 27, and `c` has value `'*'`, then

The call:	**will print:**
`printf("Hello\n");`	`Hello`
`printf("%d+%d=%d\n",x,y,x+y);`	`3+27=30`
`printf("x+%d=%d\n",5,x+5);`	`x+5=8`
`printf("%cc%c\n",c,c);`	`*c*`

Note the use of `\n` (*newline*) to indicate that the line printed is to be terminated by a line feed. Several such *escape characters* are available, namely:

`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\'`	single quote
`\"`	double quote
`\\`	backslash
`\0`	ASCII null (0)

`scanf` is very similar in form to `printf`. For the time being we will restrict ourselves to two forms, namely to read single integer and character variables respectively. The simplest such forms are:

> `scanf("%d",&`*integer_variable_name*`);`
> `scanf("%c",&`*character_variable_name*`);`

The use of the ampersand (`&`) before the variable names will become clear later.

Let us consider a simple program to illustrate these concepts:

```
#include <stdio.h>
/* Square/Cube Example 1 */

main()
{
   int x, x_sq;

   printf("Please enter a number\n");
   scanf("%d",&x);
   x_sq = x*x;
   printf("%d squared is %d, cubed is %d\n", x, x_sq, x*x_sq);
}
```

All input and output is done from and to files. The standard I/O library provides three special files, called `stdin`, `stdout` and `stderr` (standard input, output, and error files, respectively). Under normal circumstances the `stdin` file is the keyboard, and `stdout` and `stderr` are the screen. `printf` always prints to `stdout`, while `scanf` reads from `stdin`. `stderr` is used for error messages.

UNIX allows us to read from and write to data files, simply by using *I/O redirection* and `stdout`. When we execute a compiled program (by typing its name) we can specify whether to redirect the source of the input, or redirect the destination of the output, as follows:

> *file*	Create *file* and use it for **stdout** output.
< *file*	Take **stdin** input from *file*.
>> *file*	Append **stdout** output onto the end of *file*.

Thus, if we wanted `printf` to print to a file called **progout** and `scanf` to read from a file called **progin**, we could for example use:

```
a.out <progin >progout
```

If we wanted to send the output to both the screen and the file, we could use the UNIX tee, as follows:

```
a.out <progin | tee progout
```

2.3.4 Passing Arguments to Functions

Our use of `printf` above illustrates calling a function with arguments. To define a function with arguments in C, we must list the argument names, in order, in the *formal parameter list* (parameter=argument) which follows the function name. This must immediately be followed by the type declarations of the arguments. The order of the arguments in the argument list is important, as the same order must be used when calling the function, but the order of the declarations is unimportant. For example, say we wanted to print the squares and cubes of the first five integers. We could use:

```
/* Square/Cube Example 2 */

main()
{
   doline(1); doline(2); doline(3); doline(4); doline(5);
}

doline(num)
   int num;
{
   printf("%d\t%d\t%d\n", num, num*num, num*num*num );
}
```

(Note: from now on, we will omit the `#include <stdio.h>` from our examples, and assume it is always present).

All arguments in C are *passed by value.* In other words, only the value of the argument is passed to the function. Upon entry to the function, space is allocated for the formal parameters, and these are initialised with the passed values. Using a variable as an argument in a function call will not result in the variable's value being changed, even if the formal argument changes within the function. Thus, executing:

```
main()
{
   int x;
   x = 5;
   addone(x);
   printf("%d\n",x);
}

addone(n)
   int n;
{
   n = n+1;
}
```

will print the result 5, not 6. When the function is exited, the values of the formal arguments are simply discarded.

2.3.5 Returning Values from Functions

We saw earlier how to use preprocessor macros to implement simple functions, such as squaring and cubing numbers. We can do this with functions, as follows:

```
/* Square/Cube Example 3 */

square(n)
   int n;
{
   return n*n;
}

cube(n)
   int n;
{
   return n*n*n;
}
```

The `return` statement causes the containing function to be exited, and the function call effectively to be “replaced” by the returned value. We could call the above functions in different ways, for example:

```
x_sq = square(x);
printf("%d cubed is %d\n", n, cube(n));
square(2);
```

In the last example, the function will be called, and the value 4 will be returned. However, as we are not using the value returned by the function, this value will simply be discarded. It turns out that this may happen often. Unless told otherwise, C expects all functions to return an integer, and reserves space for this. This is fine when we are returning an integer, but there are two other cases: we may not want to return any value, or we may want to return a type other than integer.

These problems are easily solved: we simply have to declare the type of the function. For example:

```
#define PI 3.1415927

float circumf(radius)
   float radius;
{
   return 2*PI*radius;
}
```

C provides a special type, `void`, to solve the first problem. A `void` function is one which returns no value. For example, our `doline` function could be written:

```
void doline(n)
   int n;
{...}
```

The advantages of using `void` are firstly that you are explicitly stating that no return value can be expected from the function, and secondly, you save time and space by not allocating any unnecessary space and returning arbitrary values (which is what happens if you don’t specify `void` and don’t return a value).

A limitation of `return` is that you can only return a single value from a function. Some languages allow you to call a function and pass the arguments *by reference*, that is, passing the addresses of variables rather than just their values. This allows the function to affect the variable values directly. While all C calls are by value, there is a way to effectively make calls by reference, which we will examine shortly.

2.4 Pointers and Addresses

Memory in the computer is accessed much like post boxes in a street: each unit of memory (usually a byte made up of eight bits) has a unique numeric *address* which can be used to access it. These addresses run sequentially, so, for example, if the string `"cat"` was in memory, the address of the byte containing the `'a'` would be one more or less than the address of the byte containing the `'c'`.

Most high level languages hide this level of memory access, an exception being BASIC's **peek** and **poke**. C, however, provides an elegant mechanism for this task.

We can find the address of any variable or function by using the *address-of* (`&`) operator, and we can access the contents of any address by using the *contents-of* (`*`) operator. These two operations are complementary, and in fact the compiler removes the combination `*&` wherever it finds it. The reverse combination, `&*`, is meaningless, as one will then be attempting to find the address of a value, and not of a variable; nonetheless, the compiler removes this combination too.

These operations open up a whole new world of types in C. For any type *t*, we now have the type *address of object of type t*. Such types are called *pointer types*, as they effectively point to objects of other types.

Declaring a pointer to a type *t* in C is easy: we simply declare the variable as if it were of type *t*, but prefix its name with a `*` to indicate that it is a pointer. The following example, though simple, will illustrate basic pointer usage:

```
main()
{
   int n,     /* an integer */
       *p;    /* a pointer to an integer */
   n = 2;
   p = &n;    /* p now contains the address of n */
```

```
    printf("%d\n",*p);
    *p = 3;
    printf("%d\n",n);
}
```

You should be able to see that the results this will produce are:

```
2
3
```

The *contents-of* operator has a higher precedence than multiplication; thus, for example, the expression:

```
a**b**c
```

is grouped as:

```
a*(*b)*(*c)
```

We will now demonstrate one of the primary applications of pointers in C.

2.5 Passing Arguments by Reference

The main advantage of pointers is that they allow us to alter a variable without knowing what the variable is, just by knowing a value, namely, the address of the variable. This means we can effectively pass variables by reference. Say we want to pass a variable of type t by reference. Instead of passing the variable, we pass its address. For the corresponding formal argument, we declare a *pointer to t*. Within the called function, we can now happily access the variable (as the object pointed to by our pointer argument). Let's see an example: suppose we want a function to return both the square and the cube of a number, in two different variables. The following program illustrates this:

```
/* Square/Cube Example 4 */

void calculate(x,xs,xc)
    int x, *xs, *xc;
{
    *xs = x*x;
    *xc = (*xs)*x;
}
```

```
main()
{
   int n, n_sq, n_cb;

   scanf("%d",&n);
   calculate(n, &n_sq, &n_cb);
   printf("%d\t%d\t%d\n", n, n_sq, n_cb);
}
```

This explains why in calls to `scanf` we send the address of the variable to read and not the value.

It should be clear that although we are effectively passing arguments by reference, we are in fact simply passing the *addresses* of these arguments by *value*. In other words, the "arguments" that are being passed by reference are not actually the arguments being passed at all. Remember, in C all arguments are passed by value; there are no exceptions to this rule. Thus we are simply achieving the effect of passing arguments by reference; we cannot really do so at all.

Pointers are an intrinsic part of C, as will become increasingly apparent. A sound understanding of them is a key to mastering the language.

2.6 More about Basic Types

So far we have only seen the basic C types such as `int`, `float` and `char`. We will now introduce some further extensions to these basic types.

`ints` are signed integers, usually represented with *2's-complement representation*. The actual range of integers allowed varies from machine to machine: on some, 16 bits are used to represent an integer, whereas others use 32 bits. Most C compilers use 16-bit integers. However, C allows one to specify the size of an integer variable explicitly with the two type specifiers

`short int`	16-bit signed integer, and
`long int`	32-bit signed integer

In each case, the keyword `int` is optional.

Similarly, a `float` is usually represented by 32 bits. Should we wish to use double precision (64-bit) arithmetic, the type `double` is provided.

Often we only need unsigned integers (for example, in an execution profile). While an n-bit *signed* integer allows a numeric range of $(-2^{n-1}, 2^{n-1} - 1)$, an n-bit *unsigned* integer allows $(0, 2^n - 1)$. To specify an unsigned type in C, we prefix the type with the keyword `unsigned`, as in:

```
    unsigned short x;
    static unsigned float root;
```

Care should be taken when mixing signed and unsigned types. The advantage of 2's complement representation is that all arithmetic can be performed as though it is unsigned. The problem that this introduces though, is that overflows may occur that will not be reported. If you are expecting output of negative numbers and you get positive ones instead, or vice versa, you have probably mixed signed and unsigned types in an overflow situation.

2.7 Characters and String Constants

We have seen that C provides a character type `char`, and that string constants in `printf` and `scanf` calls are written in double quotes. C has a novel way of handling strings, which is confusing at first but becomes very sensible once you properly understand it.

To fully understand strings, we must discuss arrays, as a string is, essentially, an array of characters. However, as mentioned earlier, pointers figure greatly in C, and we shall soon see that arrays and pointers are almost identical in C. Thus we can introduce the basics of C strings using the pointer concept.

When C detects a single quote, it knows that it is about to process a character constant. The character following the quote is the constant, and its ASCII (or possibly other) representation is used. In effect, then, character constants in C are actually just very short (8-bit) integers. In fact, when we read or print a character, it is only the `%c` format specifier that tells C it must print or read a character and not an ASCII value. A consequence of this is that we can, for example, quite happily perform arithmetic with C character variables. In fact, this is commonly done – for example, if we wished to convert an integer variable `num` with a value from 0 to 9 into a character variable `ch` representing the corresponding ASCII character, we could use:

```
    ch = num + '0';
```

Warning: If your system does not use ASCII representation, you may not be able to do this, as it is based on the assumption that character encodings of digits and letters run sequentially.

The following short program illustrates this idea:

```
main()
{
   char c;

   printf("Please enter a lower case letter\n");
   scanf("%c",&c);
   printf("Char %c has representation %d\n", c, c);
   printf("Place in alphabet is %d\n", c-'a'+1);
   printf("In upper case it is %c\n", c-'a'+'A');
}
```

Say we enter `'f'`. This has ASCII representation 102. The first `printf` will thus print `Char f has representation 102`. Now, `'a'` has representation 97; thus the expression `c-'a'+1` has value 6, which is the position of `f` in the alphabet. Adding `'A'` (ASCII 65) to `c-'a'` (5) gives the result 70, which is the ASCII code for `F`.

To illustrate a simple example of *strings*, consider the call:

```
printf("Hello\n");
```

When C encounters a double quote, it knows it has found a *string constant* (a string being a sequence of zero or more characters). During compilation, all the string constants in a program are put in the **.data** section of the file (recall, in UNIX, the **.data** section holds the initialised data from your program). Within the program, the string is replaced by its address. Thus, *a string constant in C is converted to a pointer to char.* In fact, all strings in C are treated as pointers to char. Furthermore, the first argument of a `printf` or `scanf` call must be a pointer to char. Thus we could write:

```
char *i,*o,ch;

i = "Please enter a character\n";
o = "%c";
printf(i);
scanf(o,&ch);
```

C strings are sequences of characters which are terminated by the ASCII character NUL (that is, a byte with value zero - we will refer to this character as the null character). This null is important, as it is the only way C identifies the end of the string. When using string constants, as we have been until now, the null byte is inserted automatically by the compiler.

We can extend a string constant over more than one line in exactly the same way as we did with preprocessor macros; that is, we end all but the last line with a backslash (\).

Another point to note is that C pointers are usually similar to integers. Thus, if we were to print `i` or `o` above as numbers, we would get the addresses at which these two string constants are stored. We can print the strings pointed to by using `%s` in `printf` statements, for example:

```
printf("Address is %d, String at address is %s\n",i,i);
```

`%s` is also used for reading strings with `scanf`. Note that the argument corresponding to a `%s` must in either case be a pointer to char, representing the start address of a null terminated string of characters.

2.8 Numeric Constants and Character Escapes

Numeric constants in C are not restricted to decimal numbers: we can also use *hexadecimal* (base 16) and *octal* (base 8) numbers. Octal numbers are indicated by a zero (`0`) prefix and hexadecimal numbers by a `0x` prefix. Thus, the following represent the same numbers:

`16` (decimal) `020` (octal) `0x10` (hexadecimal)

An integer constant can be followed immediately by an `l` or `L` to specify explicitly that it is of type `long`. All floating point constants are of type `double`; thus this does not apply to them.

Octal numbers are used in C to represent characters as well; this is particularly useful for control codes. Within quotes, a backslash followed by a number is taken to be a *character escape*. The number is interpreted in octal, and can be at most 377 (octal). For example, many terminals emit a beep when ASCII 7 (control-G) is printed. We could implement a beep macro as:

```
#define BEEP printf("\7");
```

Floating point numbers can be expressed using either decimal or scientific notation. Some examples:

```
3.14159       3.14159e0     0.31415927e+1
31.415927e-1  0.31415927e1
```

2.9 More about `printf`

We have seen that the function `printf` takes a string argument which may contain *conversion specifiers* such as `%d` to indicate the place and format for printing other arguments. Everything in the control string which is not a valid conversion specifier will be printed. The conversion specifiers are considerably more powerful than we have seen so far.

The general form of a `printf` conversion specifier is:

`%`[`-`][`+`][ ][*fieldwidth*][`.`][*decimal_places*]*conversion_chars*

The square brackets [] indicate optional elements of the specifier, and are not part of the specifier itself. The optional `-` indicates that the result should be printed left-justified (the default is right-justified). The optional `+` indicates that a sign should always be printed, even if the number is positive. Alternatively, a space indicates that a space should be printed before positive numbers instead of a `+` sign. If neither a `+` nor a space is present, no character is printed before positive numbers.

fieldwidth specifies the field width, i.e. the *minimum* number of places to be used for printing. If the value to be printed occupies less places than *fieldwidth* specifies, spaces are printed to fill it up, otherwise the entire value is printed with no spaces. If *fieldwidth* begins with a zero, the print field is filled up with zeroes instead of spaces. A `*` can be used instead of a number, in which case the field width must be passed as an argument in `printf`. For example:

```
printf("%-*s",fw,str);
```

will print the string pointed to by `str` left justified in a field of width `fw`.

The `.` is used simply to separate the field width and decimal place numbers, and is only essential if we use a decimal place number. *decimal_places* indicates, for a `float` or `double`, the number of decimal places to be printed, or, for a string, the number of characters of the string to be printed. Thus:

```
printf("%.3s","January");
```

will print `Jan`.

conversion_chars can be any of the following:

`c`	character
`d`	signed decimal integer
`ld` or `D`	signed long decimal integer
`u`	unsigned decimal integer
`lu` or `U`	unsigned long decimal integer
`o`	unsigned octal integer
`lo` or `O`	unsigned long octal integer
`x`	unsigned hexadecimal integer
`lx` or `X`	unsigned long hexadecimal integer
`f`	decimal float or double
`e`	scientific (exponent) float or double
`g`	shortest of `e` or `f`
`s`	string

`printf` returns the number of characters printed (including the terminating null), or a negative value if an output error was encountered.

2.10 More about `scanf`

`scanf` has a format very similar to `printf`. The control string is now interpreted as specifying the format of the *input*. Any spaces, tabs or newlines in this string are ignored: C expects such characters (known as *whitespace characters*) to separate the input items anyway, so they are not needed. Any characters in the control string other than whitespace or conversion specifiers are expected to be found as well. As an example, the call:

```
scanf("%d,%d", &num1, &num2);
```

when given the input:

```
23,7
```

will return with `num1` set to 23 and `num2` set to 7. However, the input:

```
23 7
```

will result in `num1` being set to 27, after which `scanf` will find the 7 instead of the comma, and will terminate. Thus no value will have been read for `num2`. If we had left out the comma from the control string, the second case would be fine, but the first case wouldn't, for the same reason as above. `scanf` returns the number of arguments successfully read, so we can test for such cases.

Conversion specifiers in `scanf` control strings have the following form:

`%[*][`*fieldwidth*`]`*conversion_chars*

The optional `*` indicates that the specified input is simply to be skipped, that is, that there is no corresponding address in the argument list to `scanf`. This is useful for skipping over data items in the input that we are not interested in (for example, when we are reading from a file of records, rather than the keyboard, and are only interested in some fields of the records).

Here *fieldwidth* specifies the *maximum* length of the specified item. If we are reading a string with a field width of 10, for example, `scanf` would stop reading the string the moment it had successfully read 10 characters. The conversion characters for `scanf` are:

`c`	read a character
`h`	read a short integer in decimal
`d`	read an integer in decimal
`ld` or `D`	read a long integer in decimal
`o`	read an integer in octal
`lo` or `O`	read a long integer in octal
`x`	read an integer in hexadecimal
`lx` or `X`	read a long integer in hexadecimal
`f`	read a float in decimal
`lf` or `F`	read a double in decimal
`e`	read a float in scientific notation
`le` or `E`	read a double in scientific notation
`s`	read a string

Note that if we are specifying a type *t* in the control string, there must be a corresponding argument in the argument list of type *pointer to* (that is, *address of*) *t*.

2.11 Arrays and String Variables

An *array* is an ordered collection of data items (array elements) of the same type (the type of the array). A particular element is accessed by specifying the collective array name and the particular position (array index) of the element within the ordered collection. Array indices in C always start at zero.

To declare an array of type `t`, with `n` elements, we use the form:

`t` *arrayname*`[n]`

The elements of the array can then be accessed individually as:

arrayname`[0]`, *arrayname*`[1]`, `...`, *arrayname*`[n-1]`

We have already come across one type of array, namely strings. Although we only used string constants, we can have string variables as well! such variables are arrays of type `char`.

C uses an ASCII null (0) to indicate the end of a string. This is the only way that the end of a string is indicated. Thus, to store the string `"cat"` in a string variable (that is, a `char` array) would require an array size of at least 4.

2.12 Pointer and Array Equivalence

One cannot assign a string to an array, only to a pointer. Thus the following is not correct and should be rejected by most compilers:

```
char name[10];
name = "P. Smith";
```

Older compilers might not report an error, even though something disastrous may have occurred here. It is instructive to examine in detail what might happen.

An array declaration results in two actions:

- space is reserved for the array in memory, and
- the array name is changed into a constant of type *pointer to t* (where *t* is the array element type), and is set to the address of the reserved space.

In other words, the declaration of `name` in our incorrect example reserves 10 bytes of storage and associates the address of this storage with the identifier `name`. However, the assignment attempts to change the value of the pointer `name` (which is supposed to be the address of 10 bytes in the **.bss** section) to the address of the string `"P. Smith"` in the **.data** section. The only way the program has access to the 10 bytes of storage set aside in the **.bss** section, is through the address held in `name`. If this is now changed, the storage will be lost forever to our program.

The essential point here is that C converts a declaration of type *array of t* into one of a constant of type *pointer to t*. The only real difference is that an array declaration (unless it is external or a formal function argument, see later) reserves some storage. In particular, if we do not specify the array size, we have just declared a pointer; that is, the declarations `t n[];` and `t *n;` are equivalent. This equivalence extends far, as we shall see next.

2.13 String Functions

If strings can only be assigned to pointers, then how are values assigned to string variables? C provides a set of functions specifically for string manipulation in the run-time libraries. To use these functions, the header file for the string library must be included, using

```
#include <string.h>
```

For now, the most important functions are:

`strcpy(s1,s2)`
`char *s1,*s2;`
	Copies string `s2` to string `s1`.

`strlen(s1)`
`char *s1;`
	Returns the length of string `s1`.

`strcmp(s1,s2)`
`char *s1,*s2;`
	Returns 0 if the strings are equal, a negative number if `s1` is less than `s2`, and a positive number otherwise.

The last function, `strcmp` (string compare), can be used for equality tests and for alphabetic ordering. “Less” here is taken to mean “comes before, alphabetically.” Since string assignment is possible using `strcpy`, our earlier error could now be corrected with

```
char name[10];
strcpy(name,"P. Smith");
```

We can then access the letters of the name by selecting elements of the `char` array. More specifically:

```
name[0] is 'P'  name[1] is '.'  name[2] is ' '
name[3] is 'S'  name[4] is 'm'  name[5] is 'i'
name[6] is 't'  name[7] is 'h'  name[8] is '\0'
```

2.14 Pointer Arithmetic

We mentioned earlier that pointers are usually stored as integers[3]. This means we can perform arithmetic on them. However, C performs pointer arithmetic slightly different to normal arithmetic. All arithmetic operations are automatically scaled by the size of the type to which is being pointed. In fact, C provides an operator, `sizeof`, which can be applied to any variable or type name, and will return the number of bytes occupied by that variable or a variable of that type. If `n` represents an integer, and `p1` and `p2` are pointers, pointer arithmetic can thus be defined as

Expression	**Is equivalent to**
`p1+n`	`p1+n*sizeof(*p)`
`p2-n`	`p2-n*sizeof(*p)`
`p1-p2`	`(p1-p2)/sizeof(*p1)`

The last case is only defined when the two pointers are pointing at different elements of the same array. The difference is then the number of array elements between them.

Note that we apply `sizeof` to the item pointed to, and not to the pointer. This is because the size of the pointer is independent of the object being pointed to, and we are interested in the size of the object being pointed to. A similar point (no pun intended) applies to arrays – applying `sizeof` to an array name will not return the size of an array element, but rather the size of the entire array. An important point is that all `sizeof` operations are evaluated by the compiler, and not by the running program.

The use of pointer arithmetic extends the equivalence of arrays and pointers. In particular, if we declare:

```
t a[n];
```

where `t` is some type, then the following two expressions are equivalent, and either can be used:

```
*(a+i)
a[i]
```

The value `0` has a special meaning to pointers. It is often defined as the macro `NULL` in **stdio.h**, and indicates that the pointer points to nothing.

[3]The exception is when sizes of integers and pionters are different, e.g. 16-bit integers and 32-bit pointers. This is system dependent.

When a C program is run, the first few bytes from address 0 usually have a signature value assigned to them. At the end of the program, this value is checked, and if it has changed, the error "NULL pointer assignment" is reported.

2.15 Type Conversions and Pointers to Functions

We have seen that the differences between types in C are actually small: integers, pointers and arrays all have similarities, and characters can be treated as small integers. This fact means that converting between types in C is easy.

C performs many implicit type conversions itself. For example, an array declaration is converted into a pointer declaration, and a string constant is converted into a pointer to `char`. When a function is defined, a constant having the function name is actually created. This constant is of type *pointer to function returning t*, where *t* is the return type of the function. Thus, a function call in C actually consists of a constant pointer name, followed by parentheses which may contain expressions representing arguments. The effect of this is the following:

- Any arguments are evaluated and saved on the program stack.
- Execution passes to the code in the **`.text`** section being pointed to by the pointer.

Furthermore, when calling a function with arguments, `char`s and `short`s are converted to `int`s, `float`s are converted to `double`s, and arrays are converted to pointers before they are stacked and the function called. This makes argument passing much simpler for the compiler to handle.

Note that we can declare a variable of type *pointer to function returning a type t* with:

```
t (*pf)();
```

We can then assign a value (the address of a function) to this variable and call the function which is being pointed to with:

(`*pf)(`*arguments*`);`

Note the difference between the above declaration and `t pf()` and `t *pf()`. The latter two declarations both define functions as opposed to a pointer to a function. The first defines a function returning a value of type `t`, while the second defines a function returning a value of type *pointer to* `t`.

Assignment statements often cause type conversions. You can mix most of the types we have so far considered, but indiscriminate type mixing may cause errors, particularly if converting long types to shorter ones, or `float`s to `int`s.

You can specifically request C to perform a type conversion for you. This is known as *type casting*. To cast a variable to a type, we precede the variable with the desired type name in parentheses, for example:

```
float truncate(n)      /* Remove fractional part */
   float n;            /* of a float */
{
   return (float)((long)n);
}
```

2.16 More Complex Declarations

We now have sufficient tools for more complex declarations. Firstly, we can have *multi-dimensional arrays*. There is no fixed limit on the number of dimensions (although some systems may have restrictions). To declare an array of more than one dimension, we simply add the size specifiers for the other dimensions. In other words, to declare `a`, a $d_1 \times d_2 \times \ldots \times d_n$ array of type `t`, we use:

```
t a[$d_1$][$d_2$]...[$d_n$];
```

We access the element at $(i_1, i_2, \ldots, i_n)$ with

```
a[$i_1$][$i_2$]...[$i_n$]
```

The array/pointer equivalence still holds; in particular, we could also access the above element with:

```
*( *(...*( *(a + $i_1$) + $i_2$)...)  + $i_n$)
```

We have seen before that we can access variables declared in other files by using `extern` declarations. We can have any number of `extern` declarations for a variable, provided there is one and only one declaration of that variable which is the *defining declaration*; that is, there is a declaration somewhere which creates the variable and tells the compiler everything it needs to know about that variable[4]. With the exception of defining declarations, we do not have to specify the size of the first dimension of a C array, but all other dimensions must be specified. This is allowed since only the defining declaration causes storage to be allocated for the variable. Any other declaration (including formal parameters in functions) does not cause storage to be allocated; nevertheless, the dimension sizes must be known for all except the first dimension so that the compiler can tell how to access the elements of the array. The first dimension is unnecessary as C does not check whether subscripts are out of range.

To make this a bit clearer, consider the following code:

```
extern int tbl[][ENTRYSIZE]; /* Not a defining declaration */

printdown(str)
char str[];     /* Function argument */
{
   ⋮
}

int tbl[ENTRIES][ENTRYSIZE];     /* defining decl.  */
```

If we want to access the element `tbl[i][j]`, the compiler must be able to calculate the address of that element. The array is stored in the row order:

```
tbl[0][0], tbl[0][1], ..., tbl[0][ENTRYSIZE-1], tbl[1][0],
..., tbl[1][ENTRYSIZE-1], ..., tbl[ENTRIES-1][0], ...,
tbl[ENTRIES-1][ENTRYSIZE-1]
```

To access the element `table[i][j]`, the compiler evaluates:

```
tbl + (i * ENTRYSIZE + j) * sizeof(int)
```

This is the address of the required element. As can be seen, we do not need the value of `ENTRIES` to calculate it.

Note the values returned by `sizeof` in the following examples:

`sizeof(tbl)`	is `ENTRIES * ENTRYSIZE * sizeof(int)`
`sizeof(*tbl)`	is `ENTRYSIZE * sizeof(int)`
`sizeof(tbl[0])`	is `ENTRYSIZE * sizeof(int)`
`sizeof(**tbl)`	is `sizeof(int)`
`sizeof(tbl[0][0])`	is `sizeof(int)`

[4]The defining declaration must not have the keyword `extern` in it

We can form compound declarators from the array (`[]`), function (`()`) and pointer (`*`) declarators. The only restrictions are:

- `void` can only be used for functions returning `void`;
- arrays of functions are not allowed, although arrays of pointers to functions are;
- functions may not return arrays or functions.

Function and array declarators have higher precedence than pointer declarators, thus:

The declaration	**Declares:**
`float s();`	a function returning a `float`
`float *s();`	a function returning a pointer to a `float`
`float (*s)();`	a pointer to a function returning a `float`
`float *(*s)();`	a pointer to a function returning a pointer to a `float`
`char (*p[])();`	an array of pointers to functions returning `char`
`int (*pf)();`	a pointer to a function returning an `int`
`void *f();`	**illegal:** the type "pointer to `void`" is not allowed

While some of the more complex declarations are not often used, it is useful to know how to read and write them correctly.

We are now in a position where we know how to define and call functions, how to send and receive values to and from functions, and how to create many different types of variables. It is time to increase our ability to do things with these. Enter the C *operators*.

2.17 Operators

There are places in our program that we need values (for example, as arguments, or for the right-hand side of assignments). Wherever a value is needed, we can use an *expression*. For example, a function call is an expression, provided that the function returns a value. In order to have more complex expressions than just values, we can use operators. We have already seen some of C's *arithmetic operators*. The complete list is:

`a+b`	add `a` and `b`
`a-b`	subtract `b` from `a`
`a*b`	multiply `a` by `b`
`a/b`	divide `a` by `b`
`a%b`	find the remainder if `a` is divided by `b` (i.e. `a` *mod* `b`)
`-a`	negate `a` (i.e. unary minus)

C provides a set of *bitwise operators* for manipulating bit patterns in memory. The operators are:

`a<<b`	shift `a` left by `b` bits
`a>>b`	shift `a` right by `b` bits
`a | b`	find `a` INCLUSIVE-OR `b`
`a ^b`	find `a` EXCLUSIVE-OR `b`
`a & b`	find `a` AND `b`
`~a`	complement `a`

Similar to these, there are *logical operators* for combining truth values (Booleans). Recall that C regards zero as being false and any other value as true, with 1 being the canonical true. Using this convention, truth values in C may be any expression, although they will typically be the result of some comparison. The logical operators are:

`a && b`	true if both `a` and `b` are true; false otherwise
`a || b`	true if either `a` or `b` is true; false otherwise
`!a`	true if `a` is not true; false otherwise

In order to compare values there are *comparison operators*. These are:

`a > b`	true if `a` is greater than `b`; false otherwise
`a >= b`	true if `a` is greater than or equal to `b`; false otherwise
`a == b`	true if `a` equals `b`; false otherwise
`a != b`	true if `a` is not equal to `b`; false otherwise
`a <= b`	true if `a` is less than or equal to `b`; false otherwise
`a < b`	true if `a` is less than `b`; false otherwise

One should be particularly careful about the equality comparison operator `==`, as it is easy to use `=` by mistake. If this happens, an assignment is done instead of a comparison, often with disastrous results.

We have already met the *indirection operators* `*` and `&`, as well as the *cast operator*, `(`*type*`)`, and the basic assignment operator, `=`. C allows us to abbreviate a common form of assignment. The assignment *lvalue* `=` *lvalue operation expression* `;`

may be abbreviated to:

lvalue operation`=` *expression* `;`

For example, `a=a+5;` can be written as `a+=5;` An *lvalue* is an expression that refers to memory that can be examined and altered (the name comes from *l*eft *value*, as only *lvalues* are allowed on the left-hand side of assignments). Thus all variables are lvalues, as are all pointer contents (that is, expressions of the form `*p` where `p` is a pointer).

The operations which are permissible as *assignment operators* are:

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

It often happens in a program that you wish to add or subtract one from an lvalue. C provides special *increment and decrement operators*:

`++a`	preincrement: add one to a and return its new value
`--a`	predecrement: subtract one from a, returning its new value
`a++`	postincrement: add one to a but return its old value
`a--`	postdecrement: subtract one from a but return its old value

These are very useful for loops and for manipulating array indexes. For example:

```
printdown(s)
   char *s;
/* Prints a string down the screen */
{
   while( *s ) printf("%c\n",*s++);
}
```

(The `while` statement is described in Section 2.20.2. `while` loops terminate when the parenthesised expression has the value zero.) The increment and decrement operators also support C pointer arithmetic. If we apply them to pointer types, the increment or decrement is by the size of the object pointed to in bytes (i.e. by one "object"). In our example above, the `s` pointer is incremented by single characters (bytes) at a time, moving through the string until it reaches the null character, at which point the `while` loop terminates.

Beware of using the increment and decrement operators in arguments to macros. Because macros simply involves textual replacements, the increment/decrement operator might be applied more times than what is intended.

2.18 Precedence of Operators

Now that we have operators to work with, we can form complex expressions. However, we must take into account operator precedence and associativity. The *precedence* of an operator indicates how tightly it groups. When there is a choice in the order in which to apply operators, the operator with higher precedence is always performed first. For example, multiplication has a higher precedence than addition, as it is always performed first; thus the expression:

```
a+b*c
```

groups as:

```
a+(b*c)
```

When there is a choice of operators and they have the same precedence, we use the associativity to determine grouping. An operator which is *left-associative* always groups left first, while a *right-associative* operator groups right first. Thus:

```
a+b+c
```

groups as:

```
(a+b)+c
```

because addition is left-associative, while:

```
**p
```

groups as:

```
*(*p)
```

because `*` (indirection) is right-associative. A table summarising the operators, their precedences and associativities, is given in Table 2.1. The array operator `[]` and the function operator `()` have been discussed in Section 2.16, while the operators `.` and `->` (which are used for accessing the components of structures and unions) will be described later.

Table 2.1: C Operators and Precedence.

Precedence	Associativity	Operators	Description
16		identifiers, constants	
16	Left	`[]`	array subscripting
16	Left	`()`	function call
16	Left	`.`	direct selection
16	Left	`->`	indirect selection
15	Right	`++` `--`	post-inc/decrement
14	Right	`++` `--`	pre-inc/decrement
14		`sizeof`	size
14	Right	(*type*)	type cast
14	Right	`~`	bitwise NOT
14	Right	`!`	logical NOT
14	Right	`-`	arithmetic negation
14	Right	`&`	address of
14	Right	`*`	contents of
13	Left	`*` `/` `%`	multiplicative
12	Left	`+` `-`	additive
11	Left	`<<` `>>`	left and right shift
10	Left	`<=` `>=` `<` `>`	inequality
9	Left	`==` `!=`	equality/inequality
8	Left	`&`	bitwise AND
7	Left	`^`	bitwise XOR
6	Left	`|`	bitwise OR
5	Left	`&&`	logical AND
4	Left	`||`	logical OR
3	Right	`?:`	conditional
2	Right	`=` `+=` `-=` `*=`	assignment
		`/=` `%=` `&=` `^=`	
		`|=` `<<=` `>>=`	
1	Left	`,`	sequential

2.19 Expressions

An expression in C can be any one of the following:

- a numeric, character or string constant
- an identifier
- an expression in parentheses `()`
- an array element
- a function call
- a unary expression
- a binary expression
- a conditional expression
- an assignment expression
- a sequential expression
- a constant expression

There are other types; these involve more complex data types that we have not introduced yet.

A *unary expression* consists of a unary operator and a single operand. The unary operators are:

type casts	`sizeof`	`++`
`--`	`-` (unary minus)	`!` (logical NOT)
`~` (bitwise NOT)	`&` (address of)	`*` (contents of)

A *binary expression* consists of a two operands separated by a binary operator. The binary operators are:

`+` add	`-` subtract	
`*` multiply	`/` divide	`%` remainder
`<<` shift left	`>>` shift right	
`!=` not equal to	`<` less than	`>` greater than
`==` equal to	`<=` less or equal	`>=` greater or equal
`&` bitwise AND	`|` bitwise `or`	`^` bitwise XOR
`&&` logical `and`	`||` logical `or`	

A *conditional expression* has the form:

expression1 ? *expression2* : *expression3*

The first expression is evaluated; if it is true, then the second expression is used as the value of the conditional expression. If the first expression is false, the third expression is used instead. We can therefore read the conditional expression *e1*?*e2*:*e3* as: "If *e1* then *e2* else *e3*" For example, we can declare macros to return the greater and smaller of two numbers as:

```
#define min(a,b)    (a>b ?  b : a)
#define max(a,b)    (a>b ?  a : b)
```

We have used *assignment expressions* already. If you are familiar with some other language, you may wonder why we have assignment expressions, when other languages have assignment statements. The answer is that in C, an assignment is both an expression and a statement: an assignment expression returns the value of the right-hand side of the expression as the value of the whole expression. If we use the assignment expression as a statement, this value is simply discarded. The use of assignments as expressions will become more obvious later; two simple example will suffice for the time being.

Firstly, it allows us to perform multiple assignments, e.g.

```
a=b=c=-1;
```

The value of the expression `c=-1` (which also happens to assign the value -1 to the variable `c`) is -1, which is the value assigned to `b`, etc.

As a second example, consider the C run-time library function `getchar()`, which reads and returns a character from standard input. We could write the following:

```
char c;
int size;

printf("Do you have a 132 character printer?  (y/n)\n");
size = ((c=getchar())=='y') ?  132 : 80;
```

Here we call `getchar`, assign the result to `c`, and use the value assigned to `c` immediately within the conditional expression. Without assignment expressions, we would have to use:

```
c = getchar();
size = (c=='y') ?  132 : 80;
```

in place of the last line.

A *sequential expression* is a list of expressions, separated by commas. Each expression is evaluated in turn, and the value of each is discarded, except for the last expression, whose value is returned as the value of the whole sequential expression. The main use of sequential expressions is for loops; these will be explained later. Sequential expressions are also known as *comma expressions*.

A *constant expression* is an expression that can be evaluated at compile-time (that is, before the program is actually executed). These are needed for `#if` preprocessor commands, array size declarations, and in a few other places (which have yet to be dealt with). Most C operators are allowed for use in constant expressions, the main proviso being that they can be evaluated before execution. Constant expressions in `#if` commands are more restricted, as they must be able to be evaluated by the preprocessor, whereas all other constant expressions must be evaluated by the compiler (which is far more powerful and has considerably more information available to it. For example, the addresses of variables are known by the compiler but not by the preprocessor; thus if we have declared `static int x;` we can use `&x` as a constant expression in the program but not in preprocessor statements).

2.20 Statements

Statements are the most complex constructs in C other than functions (although a function body is simply a large compound statement). All statements in C, except compound statements, must end with a semicolon. Any expression may be treated as a statement; it is simply evaluated and its value discarded.

A *compound statement* or *block* consists of (optional) declarations followed by a (possibly empty) sequence of statements, all enclosed in braces {}. A compound statement may appear anywhere a statement may. Any declarations that are not `extern`al are local to the block, and cannot be accessed from outside the block.

2.20.1 The `if` Statement

An `if` statement has the form:

```
    if( expression )
```

>   *statement*
> `else`
>   *statement*

The `else` *statement* part may be omitted. The expression is evaluated; if it is true, the first statement is executed. If it is false, the statement following `else` is executed. If there is no `else` part and the expression is false, the `if` statement is skipped and the program continues with the next statement.

`if` statements can be cascaded as in:

> `if(` *expression* `)`
>   *statement*
> `else if(` *expression* `)`
>   *statement*
> `else if( ... )`
> ⋮
> `else` *statement*

An `else` always refers to the closest possible `if`. If any other interpretation is needed, braces can be used. For example

> `if(`*e1*`) if(`*e2*`)` *s1* `else` *s2*

where *e1* and *e2* are expressions, and *s1* and *s2* are statements, groups as

> `if(`*e1*`)` { `if(`*e2*`)` *s1* `else` *s2* }

If we wanted it to group as

> `if(`*e1*`)` { `if(`*e2*`)` *s1* } `else` *s2*

we would have to use the braces as shown.

2.20.2 The `while` Statement

The `while` statement is one of C's looping statements. It has the form:

> `while(` *expression* `)`
>   *statement*

The *expression* is evaluated; if it is false, the rest of the `while` statement is skipped and the program continues with the next statement. If *expression* is true, *statement* is executed, and the program continues from the start of the while statement. Thus, as long as *expression* is true, *statement* is repeatedly executed. Note that if *expression* is false right at the start, *statement* is not executed at all.

2.20.3 The `do` Statement

This is similar to the `while` statement. It has the form:

```
do
    statement
while( expression );
```

The main difference between a `do` and a `while` statement is that in a `do` statement, *statement* is executed before the *expression* is evaluated. Thus *statement* is always executed at least once, even if *expression* is false to begin with.

2.20.4 The `for` Statement

This is C's most powerful looping command. It has the form:

```
for( expression1; expression2; expression3 )
    statement
```

Each of the three expressions is optional, but the semicolons must be present. The effect of the `for` statement is as follows: Firstly, *expression1* is evaluated. This is used to set up any variables needed by the loop. Then the following process is repeatedly performed: *expression2* is evaluated; if its value is zero, the `for` statement terminates and the program continues with the next statement. Otherwise, *statement* is executed, after which *expression3* evaluated.

Thus, *expression2* is used to test for loop termination, while *expression3* is usually used to modify some variable(s) used in controlling the loop.

The `for` loop is equivalent to the following `while` loop:

```
expression1;
while( expression2 ){
    statement;
    expression3;
}
```

Let's see some examples. Firstly, the BASIC for loop:

```
FOR I=0 TO 100 STEP 10
```

could be written in C as:

```
for( i=0; i<=100; i+=10 ){
    ⋮
}
```

The C `for` is much more powerful. For one thing, sequential expressions may be used inside the `for(...)`, and furthermore, the expressions are arbitrary. For example, to loop through all the powers of two up to 1000, we could use:

```
for( i=1; i<=1000; i*=2 ){
    ⋮
}
```

Or, to read characters until a space is entered:

```
for( ; getchar()!=' '; );
```

In particular, this last example shows how any `while` loop could be done with a `for`. The `while` loop:

`while(` *expression* `)` *statement*

is the same as:

`for( ;` *expression*`; )` *statement*

Conversely, we have already seen how any `for` loop can be replaced by a `while`.

As a more sophisticated example, here is a function that tests whether two strings are equal:

```
int stringeq(s1, s2)
    char s1[], s2[];
{
    char *p1, *p2;

    for( p1=s1, p2=s2; *p1 && *p2; ++p1, ++p2){
        if( *p1 != *p2) return FALSE;
    }
    return *p1==*p2;
}
```

The first expression sets the pointer variables `p1` and `p2` to the start of the strings. The second expression is true if and only if neither of the current characters being compared is a zero; in other words, we have not yet reached the end of either string. The third expression sets the pointers to point to the next two characters for comparison.

The `if` statement is executed if the second expression is true, that is, we have not reached the end of a string. In this case, if the two characters being compared are different, the strings are not equal and we return false.

If we reach the end of one of the strings and have not yet found a mismatch, we exit the `for` loop. Now, either we have reached the end of both strings and the strings are equal, or one string is longer than the other. We can simply return the value of the comparison of the last two characters for this – at least one must be zero, so the other is either also zero (and we return true) or not (we return false).

Note that in the above example we did not have to use braces ({}) to delimit the body of the `for` loop, since it consists only of one statement. We nevertheless did so because it is good programming practice – if at a later stage we wish to add a statement to the `for` body, we do not have to remember to add the braces.

2.20.5 The `switch` Statement

This is a multiway branch based on the value of an expression, similar to BASIC's `on...goto` and Pascal's `case...of`. It has the form:

> `switch(` *expression* `)`
>   *compound_statement*

In the execution of the `switch` statement, *expression* (also known as the switch expression) is evaluated, and depending on its value, a branch is made to the appropriate *case label* in *compound_statement*. As the name implies, *compound_statement* must be a compound statement. Any statement within *compound_statement* may be labeled with a case label of the form:

> `case` *constant_expression* `:`

The type of *constant_expression* should be the same as the type *expression*, and no two *constant_expression*s in the case labels of the same `switch` statement may have the same value. If the switch expression has the same value as the *constant_expression* of some case label, then the compound statement is entered at that point. If no match is found, the compound statement is skipped and the program continues from the next statement.

There is a special case label, called `default:`, which will match any value that is not matched by some other case label.

As an example, consider:

```
void triangle( height )
   unsigned short height;
{
   switch( height ){
      case 4:  printf(" *******\n");
      case 3:  printf("  *****\n");
      case 2:  printf("   ***\n");
      default:  printf("    *\n");
   }
}
```

which will print

```
*****
 ***
  *
```

when called with `triangle(3)` and

```
***
 *
```

when called with `triangle(2)`. A call to `triangle` with any other parameter value than 2, 3, or 4, will cause a branch to the `default` label and a single `*` will be printed.

More than one case label may refer to the same statement in the compound statement, for example:

```
avoid quack_shrink()
{
   char c;

   printf("Do you like yourself?\n");
   c=getchar();
   switch( c ){
   case 'y':
   case 'Y':
      printf("You're cured!\n");
      break;
   case 'n':
   case 'N':
      printf("Don't be so neurotic!\n");
   }
```

```
      printf("Don't forget to leave my fifty bucks.  Come again!\n");
   }
```

(There is an error in the above program, can you spot it?) The `break` statement is described below.

2.20.6 break, continue, goto and exit()

A `break` statement causes the execution of the smallest containing `while`, `do`, `for` or `switch` statement to be terminated. Execution continues with the next statement after the `while`, `do`, `for`, or `switch`.

A `continue` statement causes the execution of the smallest containing `while`, `do` or `for` loop's *statement* to be terminated. Execution resumes at the start of the loop. The use of `continue` is discouraged, as it is often a sign of a poorly structured program.

An example of using of `break` and `continue` is:

```
for( i=0; i<5; ++i ){
   printf("At start of loop; i=%d\n",i);
   if (i==3) break;
   if (i==1) continue;
   printf("At end of loop; i=%d\n",i);
}
printf("Out of loop - i=%d\n",i);
```

This will result in the printout:

```
At start of loop; i=0
At end of loop; i=0
At start of loop; i=1
At start of loop; i=2
At end of loop; i=2
At start of loop; i=3
Out of loop - i=3
```

A `goto` statement can be used to transfer control from any statement in a function to any other statement in the function. The general form is:

goto *identifier*;

The destination statement must be preceded by *identifier* followed by `:`. The use of `goto` is strongly discouraged, with one exception – if some kind of error condition occurs in a function or a multiple nested construct, it is acceptable `goto` the end of the function or to use `goto` to break out of the nested construct. For example:

```
funk()
{
   if( ... ){
      ⋮
      while( ... ){
         ⋮
         if( error ) goto disaster;
         ⋮
      }
      ⋮
   }
   ⋮
   return;
disaster:
   printf("Error in funk\n");
}
```

The appearance of a label is taken as the declaration of the label; thus there is no need to explicitly declare labels.

C provides a system call called `exit(`*n*`)`. The execution of an `exit` call terminates the program, and returns the argument value *n*. This is usually 0 if the program terminated satisfactorily, or some non-zero error code otherwise. The standard UNIX error codes are contained in the include file **errno.h**.

2.20.7 The null Statement

This consists of just a semicolon. It is useful in two situations: when we have a loop which needs no *statement* following (as in some of our loop examples above), and for allowing labels for `goto`s at the end of blocks (for example, if we didn't want an error message in `funk` above).

2.21 More About Declarations

The *scope* of a declaration is the region of the program over which the declaration has effect. C declarations may have one of five scopes:

- *top-level declarations* have a scope extending from the declaration to the end of the program, even if the program is split up into several files (but see below);
- *arguments* have their function bodies as their scope;
- *block declarations* have a scope extending to the end of the block;
- *labels* have their containing function bodies as their scope;
- *macros* have scope from their declarations until the end of the program or an `#undef` command for the same macro, whichever comes first.

The scope of every declaration is limited to the file containing it, unless it is a global (`extern`, or non-static top-level) declaration.

A declaration may be hidden by other declarations whose scope overlaps that of the first declaration. For example:

```
int x;

main()
{
   int x;
   ⋮
   {
      int x;
      ⋮
   }
}
```

Within the function, the top-level declaration of `x` is hidden by the declaration at the start of `main()`. Within the sub-block, both of the outer declarations are hidden.

When an identifier is used before it is declared, it is known as a *forward reference*. C allows labels to be forward referenced (as well as structure, union and enumeration tags, see later). Referring to a function before it is declared is acceptable, provided the function returns an integer. If it returns any other type, it must have a *forward declaration* to enable the compiler to know what return type to expect. For example:

```
    float area(), volume();
```

would tell the compiler that the two functions `area` and `volume` each return a `float`, not an integer.

Identifier names in C can be *overloaded* (that is, the same name can be used for different objects) provided they are in different *overloading classes* . There are five overloading classes:

- *preprocessor macro names* – these are independent of any other names in the C program, as preprocessing occurs first and in a separate pass. This overloading class is an exception – all occurrences in the program of macro names will be replaced. Thus macro names cannot be overloaded.
- *statement labels*
- *structure, union* and *enumeration tags* (see later)
- *structure* and *union component names*
- all *other names* fall into an overloading class that includes variables, functions, `typedef` names (see later) and enumeration constants (see later).

It is illegal to make two declarations of the same name in the same overloading class in the same block or at the top level; we call such declarations *conflicting declarations* as the compiler cannot determine which declaration should be used. The only exception is `extern` declarations, which can be used freely, provided that there is one and only one defining declaration somewhere for each externally declared identifier. For example, our functions `area` and `volume` above have not been defined although they have been declared; thus there must still be defining declarations for them somewhere in the program, and these defining declarations would not conflict with the `extern` declarations.

At most one *storage class specifier* may appear in a declaration. The available storage classes are:

- `auto`
- `extern`

- `static`
- `register`
- `typedef`

We have already seen the first three. A `register` declaration is useful when a particular variable is used heavily in a function which needs to be optimised for speed (for example, a sort routine). The `register` storage class tells the compiler to allocate one of the computer's hardware registers for that variable, if possible. Some compilers simply ignore this storage class. You may not use the *address-of* operator (`&`) with register variables.

A `typedef` declaration simply declares a name as a synonym for a type. It looks just like a normal declaration, but the "variable" being declared is actually just a name which can be used to declare other variables of that type. For example:

```
typedef int *pointer,
            (*PFint)();
```

declares `pointer` to be a name for the type *pointer to int*, and `PFint` to be a name for the type *pointer to function returning int*. Once a `typedef` name has been declared, it may be used wherever a type specifier is needed; thus:

```
pointer a, b;
PFint   f1, f2;
```

declares `a` and `b` to be *pointers to int*, and `f1` and `f2` to be *pointers to functions returning int*.

2.22 Initialisation Within Declarations

When we declare a variable, we cannot assume anything about its initial value. Some compilers initialise all declared variables to zero, while others simply allocate storage, regardless of what values may be in that storage at the time. However, when we declare a variable, we can initialise it immediately using

type identifier = *initialisation*

Initialisers for `static` variables must be constant expressions, as these variables are initialised by the compiler (and placed in the **.data** section). Any expression may be used for `auto` variables, as these are evaluated every time the automatic variables are created. Formal arguments of functions may not be initialised.

As some compilers cannot perform floating point arithmetic, `static float`s should be initialised with numbers only (not arbitrary constant expressions).

Some examples of `int`, `char` and `float` initialisers are:

```
static int count=4*200, sum=0;

main()
{
   char ch=getchar();
   ⋮
}

static void calculate(k)
   double k;
{
   static double epsilon=1.0e-6;
   auto float leeway = k*epsilon;    /* illegal in some compilers */
   ⋮
}
```

Initialisers for pointers must evaluate to an integer, or an address plus or minus an integer constant. For example:

```
#define NULL 0
double *dp = NULL;
long *population = (long *)32000000;

char buffer[100];
char *bufptr = buffer,
     *bufend = &buffer[99];

static short s;
short *sp1 = &s,
      *sp2 = &s+3;     /* a valid initialisation, */
                       /* but not recommended */
```

```
char *greet = "Press <return> to begin\n";

float powers_of_pi[8];
float *pi = &powers_of_pi[1],
      *pi_sq = &powers_of_pi[2];
```

Arrays can be initialised as well. The last subscript of the array forms the "most tight" group. For example,

```
int a[5] = { 0, 1, 2, 3, 4 };

int b[2][3] = {
   {10, 20, 30},
   {40, 50, 60}
}
int table[2][3][4] = {
   { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },
   { {11,10,9,8}, {7,6,5,4}, {3,2,1,0} }
};
```

If the number of initialisers is less than the number of array elements, the remaining elements are initialised to zero. We also do not need to specify the size of the array, in which case it is worked out from the initialiser. For example:

```
char *names[] = {"John", "Mary", "Bob"};
```

will declare a three-element array of pointers to chars.

When initialising `char` arrays, remember that space must also be reserved for the terminating `'\0'`. For example,

```
char word[] = {'c','a','t','\0'};
```

Recall it is not correct to say

```
char word[] = {"cat"};
```

as `"cat"` has type *pointer to char*, whereas `word` is an array whose elements are of type `char`.

Finally, the following examples are all equivalent ways of declaring and initialising a 4-`char` array to contain a string, although some are clearer than others:

```
char str[4] = { 'U', 'C', 'T' };
char str[]  = { 'U', 'C', 'T', '\0' };
char str[]  = "UCT";
char str[]  = { "UCT" };
```

2.23 Structures

A *structure* is a collection of named components of various types. They are thus similar to records in a data file. For example,

```
struct{
   char *name;
   int dept;
   long tel;
} staff[50];
```

defines `staff` as an array with 50 elements, where each element consists of an employee name, department number, and telephone number.

Each time we define a structure type, we introduce a new type different from all others. If we wish to declare more than a few structures of a particular type, we must name the structure type so that we can refer to it again. We can do this using a `typedef` , for example,

```
typedef struct{
   char *name;
   int dept;
   long tel;
} an_employee;

an_employee staff[50];
```

Alternatively we can name the structure with a *structure tag.* For example:

```
struct employee {
   char *name;
   int dept;
   long tel;
};

struct employee staff[50];
```

gives the name `employee` to the structure we used earlier, and then declares an array of 50 of these structures. We could also have used:

```
struct employee {
   char name;
   int dept,
   long tel;
} staff[50];
```

for the same effect. The general form of a structure definition is therefore

> `struct` *structure_tag* {
>   *structure_description*
> } *variable_list*;

where *structure_tag* and *variable_list* are optional (although it would be pointless to leave them both out).

Structures can be forward referenced provided their sizes are not required. This allows structures to refer to one another, for example:

```
struct P{ struct Q *qp; };

struct Q{ struct P *pp; };
```

To access the components of a structure , C provides the two operators: `.` (*direct component selection*) and `->` (*indirect component selection*). The first is used on structures, the second on pointers to structures. Using our `employee` example above, we can write:

```
struct employee x, *y, z;

y = z;            /* y now points to z */
x.name = "The Boss";
x.dept = 1;
(&x)->tel = 666;
y->name = "Joe Soap";
(*y).dept = 99;
```

Structures can be initialised; for example:

```
struct S{ int a; char *b; } x = {1,"abcd"};
```

declares a variable `x` with structure `S` and initialises `x.a` to `1`, and `x.b` to point to the string `"abcd"`.

Older C compilers only allow pointers to structures to be passed to and from functions, and do not allow structure assignment (this must be done on a component-by-component basis). Newer compilers, however, allow structures to be assigned to, and passed to and fro. Structures may not be compared for equality (if you want to do this, you must write your own function to do it on a component-by-component basis).

Structure components may have any type except function types. Note, however, that a structure component may have the type *pointer to function*.

Structures may not contain instances of themselves (an infinite regression!) but may contain pointers to instances of themselves. The only overloading restriction on component names is that the component names within any one particular structure definition must be distinct.

Small integer components can be packed into small spaces using *bitfields*. The component name in the definition is followed by a colon (`:`) and the number of bits to be used. Unnamed bitfields can be used for padding. The use of bitfields is likely to be non-portable; the main use is in matching some data structure exactly. As an example, a structure to match the general form of an (assembler) instruction of the Motorola MC68000 microprocessor chip is:

```
struct instr {
   int op:4,
       reg1:3,
       mode1:3,
       mode2:3,
       reg2:3;
};
```

Such a structure would occupy 16 bits, which is the length of the standard MC68000 instruction. Thus our structure would match an instruction exactly, and give us easy access to the various fields of such an instruction.

Something worth noting is that the `sizeof` operator cannot be applied to bitfields. The operator returns a size in bytes, so its application to bitfields is meaningless. Likewise it can be dangerous to apply the `&` operator to a bitfield. Arrays of bitfields are not allowed. Unions may not contain bitfields, but may contain structures which contain bitfields, so this is no real restriction.

As most 16-bit computers do not allow 16 and 32-bit data items (such as `short` and `long ints`) to be on odd-address boundaries, while bytes (that is, `chars`) may, a structure may contain holes. For example:

```
struct eg { int a; char b; int c };
```

will not be able to be packed into 5 bytes (assuming 16-bit ints), but will need 6 bytes. Thus a one byte hole will occur in the structure (this is why we must compare structures for equality component-for-component, as the contents of the holes are undefined). Thus, `sizeof` must always be used when determining the size of a structure.

Even rearranging the above definition to read:

```
struct eg { int a,c; char b; };
```

will not help, as the compiler works out the size of a structure based on the assumption that you may declare arrays of the structure. Conceptually, we can think of this as: if we stored three of these structures in memory one after the other, how much memory would the middle one have to occupy if no invalid addresses are allowed. No matter how we arrange the above example, the answer is always 6 bytes.

2.23.1 I/O with Structures

It is possible to read and write structures, unions, and so on, using the `fread` and `fwrite` library functions. However, things are not always as simple when we wish to read a file with a particular format into an array of records. Consider a data file which has a strict line layout, such as shown in the following table.

Position	Data
1-13	Employee's ID number
14-32	Employee's Name
33-40	Employee's Salary

Assume we want to read the employee's birth dates (obtained from their ID number) and names into an array of structures defined as:

```
struct BirthData {
   int day;
   int month;
   int year;
   char name[20];
};
```

One problem that may occur is with the names – they can contain whitespace. If you use `scanf` with a `%s` conversion specifier for the string, the read will terminate at the first space character found in the person's name. This is not what we want. The solution is to use the `%c` conversion specifier instead, with the code:

```
struct BirthData buf;


scanf("%2d%2d%2d%*7c%19c%*8c",
            &buf.year,&buf.month,&buf.day,buf.name);
      ⋮
```

Note however that unlike with `%s`, `scanf` does not add the terminating `'\0'` byte automatically if `%c` is used. Also, `%c` reads in and keeps all trailing white space, so that `strlen(buf.name)` would always be nineteen, regardless of the name read (of course, if we haven't added the terminationg byte ourselves, then the string may well be longer than 19 characters).

If you want to read and write complete structures (as opposed for a field-by-field method as used above), you should bear in mind that the whole structure will be written, including any 'holes'. You can use the `sizeof` operator to find out how many bytes are occupied by a structure. This is the number of bytes that will be read and written. The advantage of reading and writing whole structures is that we are guaranteed that the field will be read back in correctly, as a specific number of bytes will be written/read at a time. You should open the file in binary mode, not text mode, to ensure that this works correctly. Also bear in mind that structures which contain pointers will be read in with the old pointer values – if the memory allocated for the structures that are read in is not at precisely the same address as the structures that were written out, these pointer fields will no longer be valid.

The following function and macros are useful for copying whole structures. They are independent of the structure to be copied:

```
void _scopy(dest, source, len)
   char *dest, *source;
   int len;
{
   if( len > 0 ){
      while( len-- ){
         dest++ = *source++;
      }
   }
}

#define structcopy(d,s)         _scopy(&d,&s,sizeof(s))
#define pstructcopy(d,s)        _scopy(d,s,sizeof(*s))
```

`structcopy` takes structures as arguments, while `pstructcopy` takes pointers to structures as arguments.

2.23.2 Structures and Absolute Addresses

Structures can be a convenient way to access tables at fixed locations in memory. For example, IBM PC's have a table called the disk base table at address `0xFEFC7`. We can define a structure with fields corresponding to this table as:

```
struct{
   char specify1,
        specify2,
        wait_time,
        sector_size,
        last_sector,
        RW_gap_length,
        data_length,
        format_gap_length,
        format_value,
        settle_time,
        startup_time;
} far *DskBaseTbl = 0xFEFC7;
```

The actual details of this table, as well as the PC-specific keyword `far`, is beyond the scope of this discussion. Having created our table structure, we can now access the elements of the table using the usual structure pointer operations. (Hackers, take note – the address of the table as given, is the default table in the ROM BIOS; thus you cannot change any of these values. The location of the RAM version of the table is given by interrupt vector 1E).

2.24 Unions

Much of what we have said about structures also applies to the type `union`. Unions too have a number of components, and have declarations which are (superficially, at least) the same as structure declarations, except using the keyword `union` instead of `struct`. However, a union can only store one of its components at a time; in other words, the components are overlaid in memory. This allows us to access areas of memory in different ways. In other words, a C `union` is similar to a Pascal variant record.

C does not provide a way to determine which component of a union was last assigned; it is up to the programmer to keep track of this. This is unlike

Pascal variant records, which have a tag to indicate which type is present. This can easily be done in C as well; C just doesn't demand it (thus allowing greater flexibility). For example,

```
#define STRING 0
#define VALUE 1
#define CHR 2

struct reply {
   short tag;
   union {
      char str[8];
      long val;
      char ch;
   } is;
} answer;
```

defines a variable, `answer`, to have the `reply` structure. The latter consists of two fields, `tag`, and `is`, where `is` can be any of three things: a `char` array, a `long`, or a single character.

To assign to this union, we can use:

```
answer.tag = STRING;
strcpy( answer.is.str, "Yes");
```

or

```
answer.tag = VALUE;
answer.is.val = 43;
```

or

```
answer.tag = CHR;
answer.is.ch = 'N';
```

The advantage of using tags is obvious: we can now determine what type of data the union is holding by examining `answer.tag`. We can then easily write functions such as:

```
print_answer( a )
   struct reply *a;
{
   switch( a->tag ){
      case STRING:
         printf("%s\n", a->is.str);
         break;
      case VALUE:
         printf("%ld\n",a->is.val);
         break;
      case CHR:
         printf("%c\n", a->is.ch );
         break;
      default:
         printf("ERROR - invalid reply tag!\n");
      }
   }
```

Bit fields are not allowed in unions, and unions cannot be initialised. In other respects, unions and structs are very similar.

Note that structures and unions cannot be cast to other types. However, pointers to structures and unions can be cast to pointers of other types (including to structures and unions of other types). Thus the following code is legal (although it is pointless and dangerous):

```
   struct A { int i; };
   struct B { float f; };

   void jam_float_into_int(val,a)
      float val;
      struct A a;
   {
      struct B *b = (struct B *)&a;
      b->f = val;
   }
```

2.25 Enumerations

Enumerations are a recent addition to C and are not supported by many older compilers. Although they are redundant, since macros can be used for precisely the same purpose (symbolic names for constants), they are more convenient and less error prone. Furthermore, symbolic debuggers give enumerations an edge over macros, as they cannot deal symbolically with macros but can with enumerations.

An enumeration is a set of values represented by identifiers known as *enumeration constants*, and like structures and unions, an enumeration declaration may include a tag name for the enumeration. For example

```
enum committee {
   president, vice_president, secretary, treasurer
} member;

member=secretary;

if( member==secretary ) printf("The Secretary\n");
```

declares an enumerated type with name `committee` and a variable of that type named `member`, assigns an enumerated constant `secretary` to the variable, and tests it for equality.

Enumerations are essentially lists of integer constants, and the compiler simply replaces references to enumeration elements with the appropriate constant value. It happens that C represents enumeration constants with integers, numbered (by default) from 0 upwards. This numbering implies an order on the constants, so that in our example

```
president < vice_president < secretary < treasurer
```

(which might reflect the real power wielded in a committee). This could have been done with macros, using

```
#define PRESIDENT      0
#define VICE_PRESIDENT 1
#define SECRETARY      2
#define TREASURER      3

unsigned short member;
```

It is possible to specify which values must be associated with enumeration constants. The rules used are that the first constant has value zero, unless otherwise specified, and all others have a value of one greater than the previous constant, unless otherwise defined. Thus if we declare:

```
enum colour{
   red=2, green, blue,
   yellow=8, black, white
};
```

The associated values will be:

```
red=2; green=3; blue=4; yellow=8; black=9; white=10;
```

2.26 Functions Revisited

A *function definition* (not to be confused with a function declaration) introduces a new function and provides some information about it, namely:

- the type of the value returned, if any;
- the types, names and number of formal parameters;
- the visibility of the function outside of the file; and
- the function body (code to be executed).

Functions may only have the storage classes `extern` (visible outside file) or `static` (not visible outside file).

The default return type is `int`. Functions may not return arrays, structs[5], or other functions, but may return any other type. For example:

```
float ( *(*fn(x,y))[] )()
⋮
```

declares `fn` as a function taking two parameters `x` and `y`, and returning a pointer to an array of pointers to functions returning `float`s.

The order of the arguments in the formal parameter list and in calls to the function must agree (although C does not actually check this); however, the order of the formal parameter declarations is unimportant. The only explicit storage class specifier allowed for formal parameters is `register` (the default is `auto`). Parameters may be of any type except `void` and "function returning..." (the latter will simply be treated as function calls, with the returned value used as the actual argument). To repeat, however, pointers to functions may be used as parameters.

A `return` statement without an expression is allowed (although if the function is not of type `void`, the returned value is unpredictable). This allows us to request a function to be exited before the end. If the function is of type `void`, it is an error to have an expression in a `return` statement. Furthermore, `void` functions may never be called in a context which requires a value. Thus:

```
extern void factor();
⋮
a = 2*factor();
```

[5]Modern C compilers might allow structs as return values.

is not allowed. In all other cases of `return`, C attempts to cast the return expression to the return type of the function; if this cannot be done, an error occurs.

2.26.1 Variable Parameter Lists

The discerning reader might be wondering by now how functions such as `printf` and `scanf` work, as they take variable numbers of parameters. C allows us to define such functions, and they can be an extremely powerful construct. The only requirement is that there must be some way of telling how many parameters there are, and what their types are. In `printf` and `scanf`, this is done by examining the format string, as it is assumed that there are as many arguments (of the apropriate types) as there are conversion specifiers.

For example, say we wish to write a function which takes several strings as parameters, and writes each one on a new line. The simplest way to specify how many strings there are is to pass the number of strings as the first parameter. The following example function will do the task:

```
#include <varargs.h>

void print_strings(va_alist)
   va_dcl     /* Note:  no semicolon */
{
   va_list args;
   int arg_count;
   char *string;

   va_start(args);
   arg_count = va_arg( args, int);
   while( arg_count-- ){
      string = va_arg( args, char* );
      puts(string);
   }
   va_end(args);
}
```

The example shown is for UNIX systems; other operating systems may use slightly different forms. The header file **varargs.h** contains the definitions of `va_dcl`, etc. The meaning of each of these is:

`va_alist` is used as the parameter list in the function header.

`va_dcl` declares `va_alist`. There should be no semicolon.

`va_list` defines the type of a special variable which is used to traverse the argument list. This variable should never be modified directly.

`va_start` initialises the special variable to point to the start of the argument list.

`va_arg` returns the next argument in the list. It takes two arguments itself, the first being the special variable described above, and the second being the type of the argument we are about to get from the list.

`va_end` cleans up any mess at the end.

2.26.2 Passing Arguments to `main()` from UNIX

As we have mentioned at the biginning, each C program must have one and only one function called `main`. UNIX allows arguments to be passed to `main()`, which can be defined as:

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

where `argc` is the argument count (the number of arguments passed), and `argv` is the argument vector, an array of pointers to strings which hold the arguments. The first of these arguments is always the name of the program. For example, say we have a program renum which renumbers BASIC programs, and we wish to be able to invoke it with

renum *input_file output_file start_number step*

In other words, to renumber the program **myprog.b** starting with line 100, using steps of 10, and sending the output to **newprog.b**, one would use:

```
renum myprog.b newprog.b 100 10
```

If we had declared the `main` function as shown earlier, we would have the following:

```
argc = 5
argv[0]="renum"       argv[3]="100"
argv[1]="myprog.b"    argv[4]="10"
argv[2]="newprog.b"
```

These arguments can be used by `main` instead of having to prompt for and read them from the keyboard. This, in fact, is how the UNIX commands such as `cc` get their arguments.

Conceptually, `main` is the function with which execution begins. In reality, all C implementations include a special piece of code, the *startup code*, where execution actually begins. This code is responsible for calling the `main` function and passing it the array of command line arguments, opening the `stdin`, `stdout` and `stderr` files, and setting up the program stack and memory heap. It usuallly also cleans up after `main` exits, and passes back any values returned by `main` or calls to `exit` to the operating system.

An example of the use of the startup code is as follows: on MS-DOS systems, unlike UNIX, the command shell **COMMAND.COM** does not expand wild characters. In other words, if we run a program prettyprint with the command line:

```
prettyprint *.c
```

then `argv[1]` will have the value "`*.c`". On a UNIX system, the shell first expands the `*.c` pattern into a list of matching files, and passes these to the program. If we wanted this to occur in MS-DOS, we would have to do the work of matching the files in our `main` function. An alternative provided by some C compilers is to have two versions of the startup code – one which does no wildcard expansion, and one which does. The latter is obviously bigger than the former, so by default the former will be used. If we need wild character expansion, then we can link the program with the alternative startup code. We then do not have to make any changes to the `main` function.

Chapter 3

The Run-Time Libraries

In the previous chapters we have examined the whole of the C language. However, as mentioned before, C provides a large number of functions in the *run-time libraries* which can be used to simplify programming tasks. Under UNIX there are numerous libraries, many of which are concerned only with UNIX system programming. We will rather focus on the five most common libraries, the *standard input/output* library, the *string* library, the *character function* library, the *storage allocation* library, and the *maths* library. To use these libraries, we must include special system header files which will give us access to them. These are:

Standard I/O	`#include <stdio.h>`
String functions	`#include <string.h>`
Character functions	`#include <ctype.h>`
Storage allocation	predeclared by C compiler, else `malloc.h`
Math functions	`#include <math.h>`

We will deal with each of these, from the simplest to the most complex.

3.1 Character Processing Functions

The character processing functions are used for classifying and converting characters. Each of these functions takes a single argument of type `char`. On some systems these functions are implemented as macros. Be careful when using them with arguments that have side-efects, for example increment/decrement operators. The most common functions are:

`isalnum(c)`
: returns true if `c` is an alphabetic or numeric character.

`isalpha(c)`
: returns true if `c` is an alphabetic character.

`isdigit(c)`
: returns true if `c` represents a decimal digit.

`islower(c)`
: returns true if `c` is a lower case letter.

`isupper(c)`
: returns true if `c` is an upper case letter.

`isprint(c)`
: returns true if `c` is printable (not a control character).

`ispunct(c)`
: returns true if `c` is a punctuation character.

`isspace(c)`
: returns true if `c` is a whitespace character (that is, a space, horizontal or vertical tab, carriage return, newline, or form feed character).

`toint(c)`
: returns the decimal value of a character representing a hexadecimal digit.

`tolower(c)`
: returns `c`, unless `c` is an upper case letter, in which case the lower case equivalent is returned.

`toupper(c)`
: returns `c`, unless `c` is a lower case letter, in which case the upper case equivalent is returned.

3.2 String Handling Functions

All of these functions assume that the strings passed as arguments are properly terminated by a null byte. In the following description, all arguments whose names begin with `s` are of type `char*` (i.e. *pointer to char*), all those beginning with `c` are `char`s, and all those beginning with `n` are `int`s.

`char *strcat(s1,s2),`
`char *strncat(s1,s2,n)`
: Append string `s2` to string `s1`, and return a pointer to the result. `strncat` appends at most `n` characters of `s2` to `s1`.

`char *strchr(s,c),`
`char *strrchr(s,c)`
: Return a pointer to the first (or last if `strrchr`) occurrence of character `c` in string `s`. A null pointer is returned if `c` is not in `s`.

`int strcmp(s1,s2),`
`int strncmp(s1,s2,n)`
: Compare the two strings and return `0` if they are equal, a negative value if `s1` precedes s2 alphabetically, or a positive value otherwise. `strncmp` compares a maximum of `n` characters.

`char *strcpy(s1,s2)`
`char *strncpy(s1,s2,n)`
: Copy string `s2` to `s1`, returning a pointer to the result. `strncmp` copies a maximum of `n` characters. Care should be taken if `s2` is longer than `s1`.

`int strlen(s)`
: Return the number of characters in `s` up to but not including the terminating null byte.

`int strpos(s,c),`
`int strrpos(s,c)`
: Return the position of the first (or last, if `strrpos`) occurrence of the character `c` in `s`. The first character in `s` has position 0.

Typical implementations of `strcmp` and `strcpy` are:

```
strcmp(s,t)
   char *s, *t;
{
   while( *s==*t ){
      if( *s==0 )return 0;
      ++s;
      ++t;
   }
   return (*s-*t);
```

```
    }

    char *strcpy(s,t);
       char *s, *t;
    {
       char *d = s;
       while( *s++=*t++ );
       return d;
    }
```

3.3 Mathematical Functions

The mathematical functions provide a large set of mathematical operations, including trigonometric and hyperbolic functions, and a random number generator. The large number of these functions, as well as their (mostly) self-evident names, makes it tedious to list them here. Consult your C/UNIX manuals (for example, the UNIX Programmer's Reference Manual volume 1 section 3) for details.

3.4 Storage Allocation Functions

It is possible to allocate and deallocate memory from the system "heap" using these functions (which are predeclared by the C compiler, and hence need no `#include` file header). The functions are:

`char *calloc(num,size)`
`unsigned num, size;`

Allocate enough memory to hold an array of `num` elements of size `size` bytes each, clear the memory to zeroes and return a pointer to the start. There is a variant `clalloc`, which takes `unsigned long` arguments, used for allocating large areas of memory.

`cfree(mem)`
`char *mem;`

Free (de-allocate) the memory pointed to by `mem`, where `mem` is a pointer returned by some previous `calloc` call.

`char *malloc(size)`
`unsigned size;`

Allocate `size` bytes of memory, and return a pointer to the start. The memory is not initialised in any way, and must be assumed to contain garbage. A `long` version, `mlalloc`, is also available.

```
free(mem)
char *mem;
```

same as `cfree`, but for memory allocated by `malloc`.

```
char *realloc(mem,size)
char *mem;
unsigned size;
```

Change the size (without damaging the contents) of the previously allocated memory region pointed to by `mem`. Return a pointer to the result. If this returned pointer is not the same as `mem`, then `mem` was not a valid argument and should not be used). There is a `long` version, `relalloc`.

3.5 Standard Input and Output

The standard input/output header file, **stdio.h**, defines some useful macros and variables for use with input and output. These include an end of file marker, `EOF`, and a new type, `FILE`, used for declaring streams. A *stream* can be a file or some other producer or consumer of data, such as a keyboard or screen. In particular, **stdio.h** declares three file pointers:

```
    extern FILE *stdin, *stdout, *stderr;
```

which represent the standard streams for use by functions such as `printf` and `scanf`, and system error messages. All of our examples so far have used **stdin** and **stdout**, which can be easily redirected by UNIX to almost any stream. This is inadequate, however, if we need more than just one file each for input and output. We can declare additional file pointers ourselves, and use them as arguments to input and output functions.

The following functions are for use with **stdin** only:

```
int getchar()
```

Return the next character (or `EOF`) from **stdin**.

```
char *gets(s)
char *s;
```

Read characters from **stdin** into the array pointed to by `s` until a newline or `EOF` is encountered. The newline is discarded and the string is terminated with a null. Return `s` if successful, otherwise null.

`int scanf(f,`*list_of_pointers*`)`
`char *f;`

Described previously. Return the number of fields that were assigned successfully.

The following functions are for use with **stdout** only:

`int putchar(c)`
`char c;`

Write the character `c` to **stdout**. Return `c` as an `int` if successful, `EOF` otherwise.

`int puts(s)`
`char *s;`

Write the string `s` to **stdout** and append a newline. Return a newline if successful, `EOF` otherwise.

`int printf(f,`*list_of_expressions*`)`
`char *f;`

Described previously. Return the number of characters actually written if successful, a negative value otherwise.

All other input and output functions take a stream as an argument. To open and close streams, we use:

```
FILE *fopen(name,mode)
char *name,*mode;

FILE *freopen(name,mode,stream)
char *name,*mode;
FILE *stream;

int fclose(stream)
FILE *stream;
```

`fopen` opens the stream called `name`, and returns a pointer to it. The `mode` specifies what the stream is to be used for; it can be any one of:

`"r"`	open an existing file for input
`"w"`	create a new file for output
`"a"`	create a new file for output or append to existing file
`"r+"`	open an existing file for input and output
`"w+"`	create a new file for input and output
`"a+"`	create a new file or append to an existing file, for input and output

`freopen` is similar, except it calls `fclose` first. In other words, it allows us to associate our stream variable with a different stream using a single call to `freopen` rather than one to `fclose` followed by one to `fopen`. If an error occurs in attempting to open a file, both `fopen` and `freopen` return a null pointer (zero).

`fclose` closes a stream. If any error occurs, it returns `EOF`, otherwise it returns zero.

To check the status of a stream, we use:

```
int feof(stream)
FILE *stream;

int ferror(stream)
FILE *stream;

void clearerr(stream)
FILE *stream;
```

`feof` checks whether the end of `stream` has been reached, and returns true if this is the case, false otherwise. `ferror` returns true if the last read or write operation to `stream` resulted in an error. **clearerr** resets any such error indication on the stream.

The input and output operations on the streams are:

```
int getc(stream)
FILE *stream;

int fgetc(stream)
FILE *stream;

int ungetc(c,stream)
char c;
FILE *stream;

char *fgets(s,n,stream)
char *s;
int n;
FILE *stream;

int fscanf(stream,f,pointer_list)
FILE *stream;
char *f;

int putc(c,stream)
```

```
char c;
FILE *stream;

int fputc(c,stream)
char c;
FILE *stream;

int fputs(s,stream)
char *s;
FILE *stream;

int fprintf(stream,f,argument_list)
FILE *stream;
char *f;

int fflush(stream)
FILE *stream;
```

`getc` and `putc` are macros, while `fgetc` and `fputc` are functions. These are the equivalent of `getchar` and `putchar` but for any stream. Similarly, `fputs` and `fgets` are the equivalent of `puts` and `gets`, but for any stream. One difference, however, is that the newline is not striped or appended as with `gets` and `puts` respectively. The same goes for `fprintf` and `fscanf`. The only "new" functions here are `ungetc` and `fflush`.

`ungetc` attempts to push the character `c` back into an input stream. Thus the next call to `getc` on that stream would return `c`. This may not be possible – if it is, `c` is returned; otherwise `EOF` is returned.

`fflush` flushes the buffers associated with a stream. Most input and output in UNIX is buffered and thus writing to a stream does not mean that the actual device or file associated with that stream will get the output immediately. For example, terminal I/O is buffered on a line-by-line basis. The terminal does not send keyboard input or print screen output until it receives a character. Say, however, we wanted to prompt for user input, and have the prompt appear immediately after our message (in other words, we don't want a newline at the end of our message). Using:

```
printf("Please enter Y or N "); c=getchar();
```

is not good enough, as the message won't be printed – no newline has occurred. We can achieve the result we want with:

```
printf("Please enter Y or N "); fflush(stdout); c=getchar();
```

When we are using files for input and output, we need additional functions to allow us to move around the file, and so on. The following functions are usually provided for this purpose:

```
int fread(buf,size,num,stream)

int fwrite(buf,size,num,stream)
char *buf;
unsigned size;
int num;
FILE *stream;

int fseek(stream,offset,type)
FILE *stream;
long offset;
int type;

long ftell(stream)
FILE *stream;

void rewind(stream)
FILE *stream;
```

`fread` and `fwrite` respectively reads and writes blocks of binary data consisting of at most `num` items of size `size` from/into the buffer pointed to by `buf`. The number of items actually read or written is returned. When reading binary blocks, we cannot use `EOF` to tell if the end of the file has been reached; we must use `feof` instead.

`fseek` seeks to a position in the specified stream, allowing random access to the stream. If the seek is successful, `fseek` returns zero. `type` specifies to what `offset` is relative; if `type` is zero, `offset` is treated as relative to the beginning of the stream, if one, it is treated as relative to the current position, and if two, as relative to the end of the file.

`ftell` returns the current position in `stream`. This can then later be used in calls to `fseek` to restore this position.

`rewind` sets a stream back to the beginning; it is equivalent to the function call `fseek(stream,0L,0)`.

3.6 Executing UNIX commands from C

In UNIX environments, it is easy to invoke other commands or programs from yours – a function `system` has been included in the runtime libraries for this. `system` takes a string argument; this argument is passed to the UNIX shell and executed, after which execution of the program continues. For example, to print the date, you can simply say:

```
    system("date");
```

All C environments should provide some way of invoking other programs and commands from within programs. If the `system` function is not provided, a related function called `exec` is probably available. Check your system documentation under *System Calls* (for example, UNIX Programmer's Reference Manual volume 1 section 2).

Chapter 4

Program Development and Style

4.1 C Style

C was designed for the development of large-scale applications based on a uniform philosophy (UNIX!). The result is that the language lends itself well to writing modular programs. Programs may be split over several files which can be separately compiled and updated, they can include text from other files, conditional compilation is allowed, and the visibility of functions and variables can be easily controlled with storage class specifiers.

This is helpful in the development of large programs. However, to successfully develop and maintain a large program, good programming style is essential. This aids readability, ease of modification, and portability. Good, clear programming style is generally much more desirable than efficiency or brevity. A normal, sane person would not, for example, want to write a program like the one in Figure 4.1.

In this section, we will examine some stylistic conventions to be used with C, as well as indicate how greater readability and portability can be achieved.

The great advantage which UNIX and C provide is the "software tool" philosophy upon which they are based. A software tool (as defined by Kernighan and Plauger in their excellent book *Software Tools*) is a program that:

- solves a general problem, not a special case, and

```
long h[4];E[80],S;t(){signal(14,t);if(S)longjmp(E,1);}c,d,l,v[]={(int)t,0,2},
w,s,I,K=0,i=276,j,k,q[276],Q[276],*n=q,*m,x=17,f[]={7,-13,-12,1,8,-11,-12,-1,9
,-1,1,12,3,-13,-12,-1,12,-1,11,1,15,-1,13,1,18,-1,1,2,0,-12,-1,11,1,-12,1,13,
10,-12,1,12,11,-12,-1,1,2,-12,-1,12,13,-12,12,13,14,-11,-1,1,4,-13,-12,12,16,-
11,-12,12,17,-13,1,-1,5,-12,12,11,6,-12,12,24};u(){for(i=11;++i<264;)if((k=q[i
])-Q[i]){Q[i]=k;if(i-++I||i%12<1)printf("\033[%d;%dH",(I=i)/12,i%12*2+28);
printf("\033[%dm  "+(K-k?0:5),k);K=k;}alarm(1);Q[263]=c=((S=1)&&!setjmp(E))?
getchar():-1;alarm(0);}G(b){for(i=4;i--;)if(q[i?b+n[i]:b])return 0;return 1;}g
(b){for(i=4;i--;q[i?x+n[i]:x]=b);}main(C,V,a)char**V,*a;{for(a=C>2?V[2]:
"jkl pq";i;i--)*n++=i<25||i%12<2?7:0;srand(getpid());system("stty raw -echo");
signal(14,t);t();puts("\033[H\033[J");for(n=f+rand()%7*4;;g(7),u(),g(0)){if(c<
0){if(G(x+12))x+=12;else{g(7);++w;for(j=0;j<252;j=12*(j/12+1))for(;q[++j];)if(
j%12==10){for(;j%12;q[j--]=0);u();for(;--j;q[j+12]=q[j]);u();}n=f+rand()%7*4;G
(x=17)||(c=a[5]);}}if(c==*a)G(--x)||++x;if(c==a[1])n=f+4**(m=n),G(x)||(n=m);if
(c==a[2])G(++x)||--x;if(c==a[3])for(;G(x+12);++w)x+=12;if(c==a[4]||c==a[5]){
printf("\033[H\033[J\033[0m%d\n",w);if(c==a[5])break;for(j=264;j--;Q[j]=0);
while(getchar()-a[4]);puts("\033[H\033[J\033[7m");}}system("stty cooked echo")
;d=popen("cat - HI|sort -rn|sed -n 1,20p>/tmp/$$;mv /tmp/$$ HI;cat HI","w");
fprintf(d,"%4d on level %1d by %s\n",w,l,getlogin());pclose(d);}
```

Figure 4.1: The winner of the 1989 obfuscated C contest. It is a valid, working C program that lets you play Tetris on UNIX – it even keeps a list of high scores!

- is so easy to use that people will use it rather than write their own.

Many of the UNIX utilities are software tools. In fact, unlike most operating systems which are large complex programs, UNIX consists of a small program called the *kernel*, which is responsible for only the most basic functions. All other operating system tasks are performed by software tools written in C (or the UNIX shell command language).

In order to write a good UNIX software tool, there are some conventions that should be followed. The tool should solve a single, general problem, to keep its use as simple as possible. Wherever possible, it should be a *filter*; that is, it should read its input from the standard input and send its output to the standard output. This means that the program can be used in conjunction with other programs and the UNIX I/O redirection commands and especially pipes, to create even more powerful tools. Some stylistic guidelines follow. For a more thorough treatment of the topic (as well as writing portable code), refer to appendices E and **??** containing recent papers on the subject.

First of all, functions should perform single logical operations wherever possible. Functions should be kept short (about 20 lines is a good length), and should be preceded by comments explaining what they do, what the arguments are for, and what is returned, if anything. Any piece of code within the function whose purpose may seem obscure should also be commented. Wherever possible, allow only one entrance to and one exit out of a function or compound statement.

Macro names should be used wherever possible to represent important constants. It is conventional in C to use upper case names for such macros. Identifier names should be a compromise between meaningfulness and brevity. Similarly, `typedef` should be used to simplify declarations.

Lines should be kept short, and indentation used to show the relative nesting of statements. Three-column indentations are recommended. Complex expressions that are often repeated should be made into functions if possible, or named using macros. For example:

```
#define DISCRIM(a,b,c)   (sqrt((b)*(b)-4*(a)*(c)))
#define QUADROOT1(a,b,c) ((-(b)-DISCRIM(a,b,c))/(2*(a)))
#define QUADROOT2(a,b,c) ((-(b)+DISCRIM(a,b,c))/(2*(a)))
```

would allow us to use the macros `QUADROOT1` and `QUADROOT2` in place of complex expressions for the roots of quadratic equations.

The number of files `#include`d, apart from the standard library .h files, should be kept small.

Closely related functions should be kept in the same file. The use of `continue`, and especially `goto`, is strongly discouraged, as it is often a sign of a badly structured function.

While these points may reduce the efficiency of programs slightly, good program and data structure design are far more important efficiency factors. The small price to pay for readable programs is well worth it.

For a software tool to be truly useful, there are some user-interface conventions which should be followed. Firstly, you should use `argc` and `argv` to enable the program to be invoked using the UNIX standard command format. This is:

program [*options*] [*files*]

Options must begin with a dash `-` and may take the form:

`-`*options*

or

`-`*option* `-`*option* ...

The first form is only allowed for options that don't take arguments.

The convention is that output from the program should be sent to **stdout**, unless the `-o`*file* option is used, which specifies which file the output should be sent to. Any error messages or diagnostics should be sent to **stderr**. This prevents interference with output that might be piped as input to some other process.

The optional *files* argument specifies a list of files to be used for input, one at a time. If no files are given, **stdin** should be used as a default. This convention allows the use of UNIX wild card characters as file arguments.

UNIX programs such as `cc` follow the convention that "no news is good news." While this can sometimes be disconcerting, it does have its merits, as no unnecessary information is produced to clutter output and prevent piping of output to other routines.

Many UNIX utilities and filters are written specifically to deal with text files. Thus, by making the input and output data from your files text wherever possible, you have access to all of these utilities.

Portability can often be improved using conditional compilation and macros. A good example of this is the `FILE` type in the **stdio.h** file. Some systems use pointers to structures for file pointers (usually pointing to a structure containing information about the file, including the current position), while others use pointers to int (usually just an index into a table of file descriptors). While these are essentially the same (as we can regard an index into a table as being an address of an element of the table), they are declared differently. To declare a file, we would use

```
struct file_info *fp;
```

in the first case and

```
int *fp;
```

for the second. Suppose we wish to write portable programs for two machines, XYZ and ABC, respectively. Assume furthermore that on XYZ `ints` are 16 bits, whereas on ABC they are 32 bits, and that we want them to be 32 bits. On XYZ we would use

```
typedef struct file_info *FILE;
```

in the **stdio.h** file, while on ABC we would use

```
typedef int *FILE;
```

We could incorporate this into our own program as follows (there is not much point to the `FILE` example, as it is done for us by **stdio.h**, but it illustrates the point):

```
#define XYZ     /* Define which machine we are on */

#ifdef XYZ
typedef int long;     /* redefine ints to be 32 bits */
typedef struct file_info *FILE;
#else
#ifdef ABC
typedef int *FILE;
#else
No machine defined!
#endif
#endif
```

The line `No machine defined!` will be rejected by the compiler, thus you will have to define the machine before you can successfully compile and run.

The preprocessor option -D*identifier* can also be used here. This option causes *identifier* to be defined to the preprocessor. Instead of explicitly defining the machine in the program, we could specify it as the -D argument to cc, for example,

```
cc -DXYZ myprog.c
```

4.2 Common Mistakes

Debugging a C program can be as easy or difficult as you make it for yourself. Well-structured, readable code simplifies the task considerably to start with. Another good practice is to develop the program bit-by-bit (say three or four functions at a time), thoroughly testing each part before moving on. C aids this process, by allowing us to put all top-level declarations in an `#include` file, and then keep all well-tested functions in one file, and the latest untested additions in a second temporary file. Once we are satisfied that the latest additions are working correctly, we can add them to the main file (using, for example, the UNIX `cat` command), and begin a new part in the temporary file.

C also provides a number of useful tools to aid the debugging process; these are described in the next section. Although it is impossible to predict likely errors in *a priori*, there are a few features of C that are often incorrectly used. These include:

- Using `=` instead of `==` in equality tests, thus performing an unwanted assignment, and returning an arbitrary value (the right-hand side of the assignment).
- Failing to terminate a statement with a semicolon; this is particularly the case just before `else` statements.
- Failing to use `break` statements within a `switch`; in most cases, we don't want to continue processing all statements to the end of the switch, but only those in the particular `case`; thus we need a `break` statement at the end of the case.
- Using a semicolon at the end of the argument list in a function definition, thus making that line look like a function call instead.
- Forgetting to use the *address of* operator (`&`) in calls to `scanf` and other functions requiring pointer arguments.

- Forgetting to close FILEs that have been opened. While this will not cause an error, some systems may not close the files or flush the buffers, leading to incomplete I/O.
- Incorrect use of pointers.
- Failing to ensure that operator precedences in expressions are correct (if in doubt, use parentheses).
- Using arguments with *side-effects* within macros. For example, if we have:

  ```
  #define toupper(c)     (c>='a' && c<='z') ? c+'A'-'a' : c
  ```

  and then call the macro with

  ```
  void copy_upper(s1, s2)
     char *s1, *s2;
  {
     while( *s1++ = toupper(*s2++) );
  }
  ```

  only every second character will be copied and, if we are unlucky, the null byte at the end of s2 will be missed with the result that the loop will not terminate properly.
- Failing to propagate changes made to the program throughout the whole program. This is particularly so with functions; if you change the parameter list of a function, make sure you change all calls to the functions as well, as C performs no checks on function calls, but simply performs them.

Bearing these points in mind could save a reasonable amount of time and effort. Of course, the best way to improve debugging skill is through practice – the more you program, the less mistakes you are likely to make in the first place, and the more experience you will have with identifying those that do occur.

Appendix A

The UNIX cc Compiler

The cc compiler is the main compiler of the UNIX system, and comprises several parts. These parts are often transparent to the user, since the cc command calls them as necessary. Nontheless, it is useful to know a little bit about the compiler and its constituent phases.

First, there is the *preprocessor*, cpp. This is a simple macroprocessor, which has a small set of commands allowing macros to be defined, sections of program to be conditionally compiled, text from other files to be included, and so on. cpp also strips comments from files. The output from cpp is passed as a text stream to the first pass of the compiler.

The compiler conceptually operates in two passes, although on most systems these actually occur together. The first pass takes the output from cpp and converts it into an intermediate file. This file typically contains assembly language statements for the higher level program code (such as loops), and a symbolic representation of the lower level expressions in the program. The second pass copies the assembly language statements to a further intermediate file, and generates assembly language for the expressions. There is also an optional optimisation phase, c2, after the second pass.

The compiler output is given to the *assembler*, as. This is the resident UNIX assembler and varies from system to system. The assembler converts the assembly language file into machine code, and saves the results in a file using the UNIX common object file format (or COFF). This format includes various headers, symbol table and relocation information, and three sections – one for the machine code (called **.text**), one for the initialised data in the program (called **.data**), and one for uninitialised data (called **.bss**).

The assembler output file is then passed to the *linker*, ld. This is the standard UNIX linker. It adds any necessary library functions to the file, and resolves references across files if the program is split over several files.

A.1 File Names

The name of the C source file should always end with a **.c** suffix. Two of the compilation phases create intermediate files with the same name as the source file, but with different suffixes, namely:

.s	the output from the second pass
.o	the output from the assembler

These intermediate files are usually destroyed, unless you specify otherwise (see later). The final executable file is always put in a file called **a.out**, unless you specify otherwise.

A.2 Invoking the Compiler

To invoke the compiler, use:

cc [*options*] *files*

files can be one or more **.c**, **.s** and/or **.o** files – cc will invoke only the appropriate compilation phase(s) for each file. Zero or more options may be specified. The main options are:

-c	Don't run the ld linker (that is, make only a **.o** file).
-g	Generate debugging information.
-p	Generate code for execution profiles.
-O	Optimise the code produced by the compiler.
-S	Don't delete the **.s** files.
-E	Run cpp only, sending output to the standard output.
-C	Stop cpp from removing comments.
-o *name*	Give the final (executable) file the name *name*.

Once a program has been compiled, assembled and linked without errors, it can be executed by simply typing its name.

A.2.1 Preprocessor Options

The preprocessor has a few options which can be sent to it either directly (if it is invoked as `cpp`), or via `cc`. The options are:

-I*dir*	Search the directory *dir* for `#include` files first.
-C	Don't remove comments.
-U*id*	Undefine *id* (equivalent to `#undef` *id*).
-D*id*[=*string*]	Define *id*. If the =*string* part is omitted, this is equivalent to `#define` *id*, otherwise it is equivalent to `#define` *id string*. If the string *string* contains spaces, it must be quoted (the quotes will be removed by `cpp`).

A.3 Controlling Compilation with simple Shell Scripts

If you often repeat the same compilation sequence after editing a file, you can save the sequence in a file and then execute it as a *shell script*. For example, suppose you want to compile and execute several programs, but you want to optimise all compilations and generate execution profiles. Thus, you are typically repeating this sequence:

```
cc -p -O -o output_file source_file
output_file
```

You can create a shell script to generalise this sequence. Firstly, use an editor to create a file (called say **mycc**), containing the following two lines:

```
cc -p -O -o $2 $1
$2
```

Then, after having saved the file, issue the UNIX command

```
chmod u+x mycc
```

which changes the file's access mode setting. It adds (+) execute (x) permission for the user (u) of the file mycc.

The $1 and $2 refer to the positional arguments which you supply to mycc. If, for example, you use:

```
mycc prog1.c prog1
```

$1 will be prog1.c and $2 will be prog1. This will compile prog1.c appropriately, leaving the output in the file prog1, and then execute prog1.

Shell scripts are considerably more powerful than this simple example illustrates, and can often be an effective aid in simplifying such routine tasks. However, a much more sophisticated way of maintaining programs is with the make utility, which is described later.

Appendix B

The vi Visual Editor

When you use vi to edit a file, it makes a copy of the file (if it exists) in its own work buffer. If you want to keep the changes which you make during a vi session, you must save the work buffer contents before you leave vi.

vi operates in two modes, *insert mode* (everything typed is treated as text to be put in the file) and *command mode* (everything typed is treated as a command to vi). Certain commands cause a switch from command to insert mode; to switch from insert mode to command mode, you simply hit the ESC key.

vi commands mostly consist of letters. The case is important; upper case and lower case commands typically represent different but related operations. In the rest of this chapter, we use the following notation:

↩	denotes the carriage return, or enter key.
↑*key*	denotes the `Ctrl` key pressed while pressing *key*.
ESC	denotes the escape key.

Note: not all the commands described in this chapter may work on your terminal exactly as described here. The outcome of some commands are dependent on the terminal settings currently in use.

B.1 Entering and Leaving vi

To enter vi, you type:

vi *list_of_filenames*

where *list_of_filenames* is optional. This will read the first file in the list (if there is one) into the work buffer and display the first screenful of lines from this file in command mode. If the file does not exist, it will be created as an empty work buffer.

If there are not enough lines to fill up the screen, vi will indicate the lines that don't exist by printing tildes `~`.

If more than one file name has been specified, you can edit the next file in the list by typing `:n`.

The most common way of leaving vi is by entering `ZZ` in command mode. This causes the contents of the work buffer to be saved back to the original file. Other commands to leave vi are the following:

`:q`↩	quit without overwriting the file, but only if the work buffer has been saved or has not been changed.
`:q!`↩	quit regardless of the state of the work buffer.
`:wq`↩	write the work buffer to the file and then quit.

The colon `:` indicates to vi that you wish to use an extended command. When you enter a colon, the cursor moves to the bottom of the screen, allowing you to enter the command. You must terminate an extended command with a ↩. In the rest of this chapter we will not show the ↩ at the end of extended commands – the `:` at the start of the commands should be enough to remind you that you have to terminate the command with a ↩.

Many vi commands are immediate commands. Such a command is not displayed and takes effect immediately – no ↩ is necessary.

The bottom line of the screen is a status line. It is used for displaying error messages, extended commands, file information, and so on.

The rest of the screen acts as a window on the file being edited, showing you the contents of part of the file.

B.2 Moving the Window and Cursor

The window movement commands are immediate commands using control codes: that is, you hold down the `Ctrl` key on the keyboard and press the appropriate letter. Basic window movement commands are:

↑`F`	move the window forward (down) by one screen
↑`D`	move the window forward (down) half a screen
↑`B`	move the window backward (up) by one screen
↑`U`	move the window backward (up) by half a screen

The cursor may be moved within the window, with the following cursor movement commands:

space	move one character to the right
backspace	move one character to the left
`0`	move to start of the current line
↩	move to the first non-blank character of the next line
`+`	same as ↩
`-`	move to the first non-blank character of the previous line
`$`	move to end of current line
line`G`	move to start of specified line (the default is the last line in the file)
`H`	move to top of screen
`M`	move to middle of screen
`L`	move to bottom of screen
`w`	move to start of next word or punctuation mark
`W`	move to start of next blank delimited word
`b`	move to start of previous word or punctuation mark
`B`	move to start of previous blank delimited word
`e`	move to end of next word or punctuation mark
`E`	move to end of next blank delimited word
`(`	move to start of previous sentence
`)`	move to start of next sentence
{	move to start of previous paragraph
}	move to start of next paragraph
`k`	move up one line
`l`	move right one character
`j`	move down one line
`h`	move left one character

The usual cursor control keys can be used for moving left or right by a single character and up or down by a single line (that is, they are treated as if they were `h`, `j`, `k` and `l`).

B.3 Moving by Context Searching

If you enter a slash `/` in command mode, the cursor moves to the status line and you can enter a search pattern followed by ↩. All lines from (but not including) the current line will then be searched for the specified pattern and the cursor will be positioned at the first string that matches the pattern. If the search reaches the end of the file, it will “wrap around” (continue from the beginning) to the starting position. The last search pattern is reused if no pattern is given, (i.e. if you just type `/`↩).

Alternatively the next occurrence of the last specified search pattern can be found by using the immediate commands:

`n`	repeat last pattern search
`N`	repeat last pattern search, but in opposite direction

To search backwards instead of forwards, use `?` instead of `/`.

B.3.1 Search Patterns

Search patterns consist of strings of characters, with the following special notations:

`^`	represents the start of a line
`$`	represents the end of a line
`\<`	matches the beginning of a word
`\>`	matches the end of a word
`.`	matches any single character
`*`	matches any sequence of zero or more characters
`[]`	allows character ranges to be specified
`\t`	represents a tab
`\b`	represents *backspace*
`\\`	represents \
`\`↩	represents a newline character
`\^`	represents a caret ^
`\:`	represents a colon :
`\$`	represents a dollar sign $
`\[`	represents a left square bracket `[`
`\*`	represents an asterisk *
`\/`	represents a slash `/`

The square brackets `[]` are used to represent a single character within a given range. The characters can be listed explicitly, or given as a range. For example, `[abcdefg]` and `[a-g]` each match a single letter in the range `a` to `g`, while `[aeiou]` matches any single vowel.

B.3.2 Matching Parentheses in Programs

A useful command for editing programs is the parenthesis match command `%`. By positioning the cursor on a parenthesis or brace, and using this command, the matching parenthesis or brace will be temporarily highlighted (or depending on your terminal type, the cursor may be positioned on the matching parenthesis/brace).

B.4 Repeating Commands

Some `vi` commands may begin with repetition count specifiers indicating how many times the command is to be performed. For example:

```
3j
```

will move down three lines.

To repeat the most recent command which caused a change to the file (that is, some form of insert or delete command), the . (period) immediate command can be used.

B.5 Inserting Text

Commands for adding text include:

`i`	insert text before cursor
`I`	insert text at beginning of line
`a`	append text after cursor
`A`	append text at end of line

These all put `vi` into insert mode. Everything that is now typed, up until the next ESC, will be treated as text.

While in insert mode, we can correct text as we enter it using

↑`H`	delete last character entered (many terminals will allow *backspace* to be used as well),
↑`W`	delete last word entered, and
`@`	delete last line entered.

These will only work on the current line being entered, and on text that has been typed since the last time insert mode was entered.

B.6 Opening and Joining Lines

To open or join lines we use:

`o`	open a new line below current line
`O`	open a new line above current line
`J`	join lines

When we open a new line, vi goes into insert mode and puts us at the start of the newly opened line. If we join lines, the specified number of lines (default 2), beginning with the current line, will be joined.

B.7 Replacing Text

To overwrite existing text, we use:

`r`	replace the character under the cursor with the next character typed
`R`	replace the characters from cursor onwards until the next ESC

Replacing of characters is done on a line by line basis, so we do not have to worry if the replacement line is longer than the original; this will not affect the next line.

B.8 Refreshing the Screen

Depending on the terminal type, and unless you specifically request it, vi does not immediately refresh the screen after deleting lines. Instead, deleted lines are replaced on the screen by `@` symbols. This is to speed the editor up a bit, particularly for slow terminals. You can, however, explicitly request vi to refresh the screen display by entering ↑`R` (some terminals use ↑`L` instead).

B.9 Deleting, Moving, Copying and Changing Text

To *delete* text, we use:

`x`	delete the character under the cursor
`X`	delete the character to the left of the cursor
`D`	delete from the cursor to the end of the line

as well as some others mentioned later.

When text is deleted, it is always placed in a buffer from where it can be retrieved. There is one unnamed buffer and 26 named buffers, `"a` to `"z`. Unless you explicitly select a named buffer, the unnamed buffer is used. To use a named buffer, the delete, move or copy command being used should begin with the buffer name.

Deleted text can be retrieved by using:

`p`	insert (put) the text from the buffer after the cursor or current line
`P`	insert the text from the buffer before the cursor or current line

To copy text to a buffer without deleting it, the yank commands are used:

`y`	copy text from cursor position
`Y`	copy text from current line

B.10 Object Specifiers

Commands such as the delete and yank commands should be followed by an *object specifier* telling `vi` what is to be deleted or copied. If the command letter is repeated, the object is the current line. In other words, the command applies to the current line. Other object specifiers are:

`c`	a single character
`w`	the next word or punctuation mark
`W`	the next word
`/`*pattern*	the next line containing *pattern*
`(`	the start of the current sentence
`)`	the end of the current sentence
{	the start of the current paragraph
}	the end of the current paragraph
`H`	the home (top) line of the screen
`L`	the last line of the screen

The last two cases may be prefixed by a number, specifying how many lines below H or above L the required line is.

The object specifiers are used with the following commands:

`c` *obj text* ESC	replace (change) the text from the cursor to the object *obj* with *text*.
`d` *obj*	delete text up to object *obj*.
`y` *obj*	copy text up to *obj* to the given buffer.
`!` *obj command*	execute the UNIX command *command*, using the text up to *obj* as input, and replace this text with the ouput of the UNIX command.

Some examples should help to illustrate these concepts:

`dw`	deletes the next word to the unnamed buffer.
`dd`	deletes the current line to the unnamed buffer.
`"ayw`	yanks (copies) the next word to the buffer `"a`.
`y/[a-z]`	yanks everything between (in including) the cursor position and the next lower-case letter to the unnamed buffer.
`5Y`	yanks the current and next four lines to the unnamed buffer.
`"ap`	puts the contents of the `"a` buffer after current position.
`!!date`	replaces the current line with the UNIX date.
`!}sort`	sorts the words to the end of the paragraph.

Many of the object specifiers also allow repetition counts, so the commands mostly have the general form:

[*buffer_name*]*command*[*count*]*object*

For example, to delete the next six words to buffer `f`, we use the command `"fd6w`.

B.11 String Substitution

We can combine the search and change commands by using the substitute command. This command requires both a search and a replacement string. The form of the command is:

`:`[*range*]`s/`*search_string*`/`*replacement_string*[`/g`]

The optional *range* specifies which lines of the file should be searched. The default is the current line, which is represented by a period (.). Ranges can specify a single line, or a start and finish line separated by commas. A line can be specified with an explicit line number, or with an expression involving the lines . (current line) and $ (last line), for example:

`.,.+4`	the current line plus the four lines that follow it.
`.,$`	everything from the current line to the end of the file.

The second example can be abbreviated by `;`. The range `1,$` (that is, the whole file), can be abbreviated by `,`.

The *search_string* is as before, including all the special characters. *replacement_string* is the string with which we wish to replace the search string. The only metacharacters allowed here are `\` (allowing special characters to be quoted), and `&`, which represents the text which matched the search string.

Usually, only the first occurrence of *search_tring* in any line in the range is substituted. To substitute every occurrence in the range, the `g` option should be used (this means *replacement_string* must be terminated by a `/` to distinguish it from the `g`). Thus, for example, to replace all occurrences of `UNIX` in a file with `the UNIX operating system`, we can use

```
1,$s/UNIX/the & Operating System/g
```

B.12 Undoing Commands

To undo changes made, we use:

`u`	undo the last change made
`U`	restore the current line to what it was before editing began

B.13 Setting vi Options

Many of vi's features may be controlled by the user. To enable, disable, set or list options, the `:set` command is used. To list all the option settings, use:

```
:set all
```

To list only those options that you have changed, use:

```
:set
```

To set an option, enter `:set` followed by the option name. To clear the option, prefix the name with cd no. The main options (plus acceptable abbreviations) are:

`autoindent`	`ai`	automatic indentation of lines
`autowrite`	`aw`	automatically save file before exit
`ignorecase`	`ic`	ignore upper/lower case in searches
`list`		print tabs as `^I` and end of line as `$`
`number`	`nu`	display line numbers
`showmatch`	`sm`	show matching brackets as the are typed, by flashing the cursor
`wrapmargin`	`wm`	enable automatic line splitting in the given zone

Examples of usage are:

`:set nolist`	don't print tabs and line feeds
`:set nu`	display line numbers

The autoindent option makes vi indent each line to the same left margin as the previous line. To "unindent" a line by one indentation, use ↑D. `showmatch` makes vi briefly highlight matching left brackets as you type right brackets, provided both are on the same screen. If there is no matching bracket, vi beeps to indicate a possible error.

B.14 Defining and Using Key Macros

It is possible to define key macros for commonly used sequences of vi commands. This is done with the `:map` command. Special characters can be entered into macros by using a ↑V prefix. For example, suppose we wish to enter a number of lines of the form:

```
table[21]="example";
```

where only the table index and string contents vary. To do this fast, we could define two macros as follows:

```
:map t A↑V↩table[
:map s A]="";↑VESChi
```

The first line assigns a macro to the key `t`. The macro starts a new line (by appending a carriage return to the current line) and then inserts `table[`, leaving vi in insert mode which will enable us to type in the desired index. The second line defines a macro for the `s` key. It appends `]="";` to the current line, goes into edit mode, positions the cursor on the second quote, and puts vi in insert mode, ready for us to type in the string. To remove a macro definition, use `:unmap` (for example, `:unmap t`).

B.15 Reading and Writing Files

There are ways of reading and writing text into and from the work buffer without exiting cmd vi.

The read command has the format:

`:r` *filename*

This reads the specified file and inserts it after the current line. To read a file into the work buffer and replace the contents of the work buffer, the edit command should be used:

`:e` *filename*

To write out text from the work buffer, the write command can be used. This has the form:

`:`[*range*]`w`[*option*][*filename*]

where the optional *range* is the same as for the substitute command, and the optional *option* can be one of cd ! or `>>`. If no file name is specified, the work buffer is written to file currently being edited.

If a range is given, only the specified portion of the work buffer will be written to the given file. If no range is specified, the entire buffer is written. vi will not overwrite an existing file if only part of the buffer is being written. To do this, you must use the overwrite option `!`. To append to a file rather than overwriting it, the `>>` option must be used.

The `:e` command can also be followed by an exclamation mark. This forces vi to obey the command even if changes made to the work buffer have not been saved. Be careful about using `!`; always try to perform a command without it first.

B.16 Advanced Search and Replace

The search and replace facility of vi is much more powerful than we indicated previously. Much of this power comes from being able to group regular expressions in the search string, and then use these groups as part of the replacement string.

The grouping facility is done using `\(` and `\)`. Within the replacement string, the groups are referred to as `\1`, `\2`, etc, in sequence.

Additional special characters in the replacement string are:

`\u`	convert the next character to upper case
\l	convert the next character to lower case
\U	turns on upper case conversion until \e, \E or the end of the replacement string is encountered.
\L	as for \U, but for lower case.

Here are some examples to clarify these concepts. To convert all occurences of “up and over” to “over and up”, we can use:

```
:s/\(up\)\( and \)\(over\)/\3\2\1/g
```

or, equivalently:

```
:s/\(up\) and \(over\)/\2 and \1/g
```

To convert all occurences of the words "unix" or "Unix" to "UNIX", we can use:

```
:s/[uU]nix/\U&/g
```

(Recall that `&` corresponds to the matched text). Similarly, to convert all occurences of the word "multics" to "Multics", we can use:

```
:s/multics/\u&/g
```

While these examples are trivial, and could as easily have be done using simple patterns, they illustrate the concepts. Real examples will appear when you are programming, and the true value of case conversion and regular expression grouping will then become apparent.

B.17 Miscellaneous

UNIX commands can be sent to the shell while in vi. To do this, prefix the UNIX command with an exclamation mark. For example, to find out the time, enter `!date`↩. See also section B.10 on Object Specifiers

To find out the name and size of the file we are editing, the `:f` or or ↑`G` command can be used.

B.18 Recovering from Crashes

vi continually saves the contents of the editing buffer in a file. If the system crashes, you can recover your file up to the last change made. vi normally sends you mail indicating that a recovery file is available. When you invoke vi the next time with that file name, you must invoke it with

vi -r *filename*

after which you can continue from where you left off. If you invoke vi without this option, the recovery file will be lost.

B.19 Installing vi Options Permanently

If you use certain `:set` options in vi every time you use the editor, you can have these options automatically set for you. To do this you must edit (or create if necessary) the file **.profile** in your login directory and add the lines:

```
EXINIT='set option_list'
export EXINIT
```

Every time you execute vi, the shell will send it these settings automatically.

Appendix C

Useful UNIX Tools

UNIX provides us with several tools to aid program development. These include

lint	a program checker
make, SCCS	program maintenance utilities
sdb, dbx	interactive symbolic debuggers
ctrace	a C trace utility
cxref	a cross-reference generator
cb	a program "beautifier"
make, SCCS	program maintenance utilities
ar	an archive (library) maintainer
size, nm	provide information on compiled files
prof	an execution profiler
time	time the execution of programs

We will examine some of these in the sections that follow.

C.1 lint – The C Program Checker

lint checks C programs for features which are likely to be bugs, non-portable, or wasteful. It provides stricter type checking than cc, and can find unreachable statements, loops not entered at the top, constant logical expressions, auto variables which are declared but never used, functions called with varying numbers of arguments, and functions returning values that are not always (if ever) used.

lint is invoked by:

lint [*options*] *files*

files is a list of files that are assumed to contain a single program, and are checked for mutual compatibility. Note that "lint" is a reserved word in lint, so the files should not use "lint" as an identifier.

There is a large number of lint options, of which we will examine only a few. Some of the main options are:

-a	Report assignments of longs to ints.
-b	Report break statements that are unreachable.
-c	Complain about non-portable type casts.
-h	Don't attempt to guess at likely mistakes.
-n	Attempt to check portability to IBM and GCOS C.
-p	Report possible portability problems.
-u	Do not complain about functions and variables that are defined and never used or vice-versa (useful for checking files containing only part of a program).
-v	Do not complain about unused function arguments.
-x	Report all `extern` declarations whose variables are never used.

The `exit` system call, and other functions that never return, are not properly handled by lint.

lint allows you to use comments within your files that will modify its behaviour from that point. Although their effects are simple (for example, shutting off strict type checking in expressions), these are likely to be used only by experienced lint users. Refer to the UNIX Programmer's Reference Manual volume 1 section 1 for details, if you are interested in using these.

C.2 make – A Program Maintenance Utility.

Large C programs are usually split over numerous files. Sometimes, some of the files are compiled and put into libraries, which are then linked with the other files. Numerous header files can also exist, with each C file `#include`-ing different sets of header files. Obviously, if a file is changed, the program must be recompiled. This may entail remaking libraries if the changed file is a library component, or recompiling several C files if the changed file is a header file. Keeping track of all the *dependencies* between files can become complex, and recompiling can be tedious. In an earlier section we saw how we could use shell scripts to control compilation, but these may not be adequate beyond a certain level of complexity. make is a tool which handles program maintenance automatically. It keeps track of file dependencies, and only performs those actions that are necessary to keep a program (or group of programs) up to date.

In order to tell make what to do, a *makefile* must be created. This file may have any name, but make will automatically use a default file if none is explicitly given. The allowed default filenames are **makefile**, **Makefile**, **s.makefile** or **s.Makefile**. The makefile specifies the dependencies between files, and the actions to be taken if a *target* is older than one of the files upon which it depends.

Each entry in the makefile is separated from others by one or more blank lines, and consists of a line specifying the target and its dependencies, followed by zero or more lines specifying the actions. Each of the action lines should begin with a TAB character. Consider a simple example. Say we have a program `f` made up of files `f1.c`, `bf f2.c` and `f3.c`, each of which uses the include file `f.h`. The file `f1.c` also uses an include file `config.h`. A sample makefile for this situation would be:

```
f:  f1.o f2.o f3.o     # Target and dependencies
   cc -o f f1.o f2.o f3.o     # Action

f1.o:  f1.c f.h config.h
   cc -c f1.c

f2.o:  f2.c f.h
   cc -c f1.c

f3.o:  f3.c f.h
   cc -c f3.c
```

Notice the use of `#` to specify comments. All text from a `#` to the end of that line is ignored by *make*.

When we run *make*, it will examine the makefile. If any of the dependent files are newer than the targets, the actions associated with them will be performed. For example, if `config.h` has just been edited, it is more recent than the target `f1.o`. `f1.c` will thus be recompiled to create a new, up-to-date version of `f1.o`. This will then make the program `f` out of date, so it will also be recompiled. After this, all the targets will be more recent than their dependencies, so *make* will exit.

If a dependency file is not found, *make* will try to create it. It does this by looking for an entry which has that file as the target. If no such entry is found, *make* will report that it cannot find the file, and does not know how to make it.

Make has a set of internal default rules that it will use if no explicit rule is given. Some of these are listed below:

Target	**Dependent on**	**Rule**
`x.o`	`x.c`	`cc -c x.c`
`x.o`	`x.s`	`as x.s`
`x.a`	`x.c`	`cc -c x.c;ar rv x.a x.o;rm -f x.o`

make has an elementary macro facility. A macro can be defined by:

```
macroname = string
```

A macro can then be used by:

```
$(macroname)
```

Macros in *make* are mostly used for altering compiler options. For example, we may want to be able to turn symbolic debugging on or off during compilations at different stages. Instead of changing each action line, we can use a macro. Our makefile will now look like:

```
CFLAGS = -g -c

f:  f1.o f2.o f3.o # Target and dependencies
   cc -o f f1.o f2.o f3.o # Action

f1.o:  f1.c f.h config.h
   cc $(CFLAGS) f1.c

f2.o:  f2.c f.h
cc $(CFLAGS) f1.c

f3.o:  f3.c f.h
cc $(CFLAGS) f3.c
```

If we wish to disable symbolic debugging, we now need only to change the macro definition.

make has several command-line options that can be specified. These include:

-s	Don't print command lines before executing them (silent mode).
-n	Print commands to be performed; don't actually do them.
-r	Do not use the built in rules.
-f*file*	Use *file* as the makefile

C.3 SCCS – The Source Code Control System

When systems are being developed, it is often useful to keep old versions of programs, for reference and also occasionally to recover a working version of a file when a line of development proves to be unsuccessful. The Source Code Control System (SCCS) is a collection of programs which allows multiple versions of programs to be stored in a space efficient way (rather than storing the programs, only the most recent versions are stored, together with the sets of changes that must be made to restore them to older versions), and to be retrieved upon command. SCCS is a sophisticated system and a description of its use is beyond the scope of this book. Interested readers are referred to the UNIX manuals for details. A good introduction to SCCS (as well as many other UNIX utilities), is the book *UNIX Utilities – A Programmers Reference*, by R S Tare (McGraw-Hill 1987).

C.4 sdb – A Symbolic Debugger

sdb is a powerful symbolic debugger for C programs. It makes use of the program source files, thus it is possible to see at any time which C statement is being processed. In order to use sdb to debug a program, the program must have been compiled with the -g option of cc.

To invoke sdb, we use:

sdb [*object_file* [*core_file* [*directory_list*]]]

where

object_file is the name of the compiled program (the default is **a.out**).

core_file is an optional core dump of the program. This is a record of the state of the program at some particular point during execution. We can generate a core dump at any time while executing a program by typing ↑\ (this also causes the program's execution to be terminated). Alternatively, if a program crashed during execution, a core dump is automatically generated by UNIX. The default name for core files is **core**.

directory_list tells sdb where to look for the program source files. If more than one directory is given, they must be separated by colons (:).

`sdb` is terminated with the command `q`.

`sdb` maintains a current file name and current line count. If *core_file* is present, the current file is set to the core file, and the current line is set to the line where the program stopped. Thus, if a program crashed, one can immediately find out where it did so.

Variable names in `sdb` are the same as in the source programs, except that variables local to a function must be written using the form *function*:*variable*. If no function name is given, the current function is used. Array and structure elements can be accessed as usual, using `[]`, `.` and `->`. One can also specify variables by their addresses.

Lines in a source program can be referred to as *file*:*number* or *function*.*number*. In either case, *number* is relative to the beginning of *file*, or the file containing *function*. If no *number* is given, the first line of the *file* or *function* is used.

`sdb` provides a large set of commands, of which we shall examine only a few. Firstly, commands to examine program source files:

`e` *function*
`e` *file*
: Sets the current file to be the specified *file*, or the file containing *function*, and sets the current line to be the first line in the *function* or *file*.

`/`*pattern*
: Search from the current line for a line containing a match of *pattern*.

`?`*pattern* Search backwards for *pattern*.

`p` Print the current line.

`z` Advance and print the next ten lines.

`w` Print the ten lines around the current line.

number Set the current line to *number*.

count`+` Advance by *count* lines, and print the new current line.

count`-` Retreat by *count* lines, and print the new current line.

The following commands allow us to examine data from the program:

`t` Print a function stack trace.

`T` Print the top line of the stack trace.

variable`/lf`

Print the value of *variable* according to the length `l` and format `f`. The length and format may be omitted, in which case `sdb` selects a length and format according to the variable's declaration. The length specifiers are

`b`	one byte
`h`	two bytes
`l`	four bytes
number	length of a string

The formats are

`c`	character
`d`	decimal
`u`	unsigned decimal
`o`	octal
`x`	hexadecimal
`f`	32-bit floating point
`g`	64-bit floating point
`s`	string – print characters from address contained in variable,
`a`	string – print characters from variable's address
`p`	pointer to function

variable`=lf`
linenumber`=lf`
number`=lf`

Print the address of *variable*, *line number* or *number* in the specified format. `l` and `f` are as before.

variable`!`*value*

Set *variable* equal to *value*

A program can be executed under `sdb`'s control. `sdb` allows one to set breakpoints at any point in the program. When an active breakpoint is encountered, `sdb` reports that it has reached the breakpoint, and stops executing the program. One can then use `sdb` freely, and continue executing the program if one wishes. The following commands allow control of the execution of a program:

count `R`
count `r` *args*
Run the program (with the arguments *args* if `r`). After using `r` once, one can omit *args*, in which case the previous *args* will be used again. The optional *count* specifies the number of breakpoints to be skipped over before making all breakpoints active.

line `C` *count*
line `c` *count*
Continue execution after a breakpoint or interrupt. *count* is as before. `c` continues, ignoring the signal that caused the program to stop. If *line* is given, a temporary breakpoint is set at that line, and deleted when the command finishes.

count `s`
count `S`
Run the program for *count* lines (the default is one). If `S`, single step through called functions as well.

function`(`*args*`)`
function`(`*args*`)/f`
Call *function* with *args*. If the second form is used, the returned value is printed with format `f`. The default format is `d`.

line `b` *requests*
Set a breakpoint at *line* (the default is the current line). If *requests* are given, then when the breakpoint is encountered the *requests* are executed and the program continues; otherwise control returns to `sdb` when the breakpoint is encountered. Multiple requests can be separated by `;`.

line `d`
Delete the breakpoint at *line*. If no *line* is given, `sdb` prints each breakpoint and asks confirmation before deleting it.

`B` Print the current breakpoints.

`D` Delete all breakpoints.

`I` Print the last line executed.

line `a`

Announce. If *line* is of the form *function*:, this is the same as *function*: `b T` else if *line* is of form *function*:*line*, it is the same as *line*`b I` The effect is to print the line or function name every time it is executed.

As in other UNIX utilitites, commands can be sent to the UNIX shell by preceding them with `!`.

C.5 dbx – A Symbolic Debugger

dbx is a symbolic debugger that is used in BSD versions of UNIX, as well as in IBM's AIX system. It is very user-friendly and easy to use.

Like sdb, dbx requires you to compile your C program with the **-g** option of **cc**. It will also make use of a core dump file if there is one (see the section on sdb for a description of what a core dump file is). Recall that you can force a core dump at any stage with ↑\.

dbx takes two optional arguments, the first being the name of the file to be debugged, and the second is the name of the core dump file. The defaults are **a.out** and **core** respectively. After starting dbx, it will print a message, and then prompt you with the prompt `(dbx)`. You can type `help` to get a screen full of help, summarising the possible commands you can use. Some (but by no means all) of these commands are described in the following paragraphs.

When dbx starts, it looks for a file called **.dbxinit**, in the current directory, and if not found, in the user's home directory. If it finds this file, it runs the commands in it. This file can be used to customise dbx to an extent, as well as to repeat a set of commands. Furthermore, the option -c*file* can be used to specify a file containing commands to be executed before reading from the keyboard.

The following operators are all valid in dbx expressions:

Algebraic:	`+ - * /` (floating) `div` (integral) `mod exp` (exponentiation)
Bitwise:	`| bitand xor ~ << >>`
Logical:	`or and not`
Comparison:	`> < <= >= != ==`
Other:	`sizeof`

The command `where` will print a *stack trace*, showing you where in your program you are. The stack trace lists, in reverse order, the different functions that are currently active, including the parameters that were passed to them. The last function listed is always the first one called, namely `main`. This command is particularly useful after a UNIX-instigated core dump, as it tells you where your program crashed.

The command `call` *func*`(`*params*`)` executes the function *func* with the specified arguments. Alternatively, `print` *func*`(`*params*`)` will do similarly, and also print the return value.

`next` [*num*] will execute the next line of the program (or *num* lines, if *num* is specified. A similar command is `step`. The difference is that if the line(s) to be executed contain a function call, `step` will stop at that call, whereas `next` will execute it.

The `run` command will execute the program. It can be followed by command line arguments for the program, as well as redirection commands for **stdin** and **stdout**. In other words, a command line can be specified as usual, except that the name of the program is omitted. For example:

```
    run -x <data >list
```

Values can be assigned to variables with the `assign` *var* `=` *expr* command, and the values of expressions can be printed with the `print` *expr*[,*expr...*] command.

The `trace` command lets you track variables as their values change. You must specify which variable to trace. When you then run the program, each time a traced variable is altered its value is printed, together with the place in the program where the value was changed. The `trace` command is extremely flexible and powerful. A similar command is `stop`. For full details, see the UNIX manuals.

To examine the program, the `list` command is used. You can specify the name of a function, or a range of lines to be listed, for example, `list 10,20`.

Finally, to exit `dbx`, we use the `quit` command.

This is a very cursory description of `dbx`'s capabilities, but it includes sufficient information to get started. Once you have the feel for these commands, look at the `dbx` entry in the UNIX manuals for full details of the entire `dbx` command set.

C.6 strip – Strip Symbolic Information

When a program is compiled with the -g option as required by sdb and dbx, the compiler adds symbolic information to the executable file that aids the symbolic debugger. This information takes up extra space which is not required once the program is debugged. UNIX provides a utility, strip, which removes the symbolic information from the file without having to recompile it. strip allows two options to control what information is stripped:

-l	Strip line number information only.
-x	Don't strip static and external symbol information.

The default action of strip is to remove all symbolic information.

C.7 ctrace – The C Trace Utility

ctrace is a utility that inserts statements into a C program to print each statement before execution, as well as to print the values of any variables referenced or modified. ctrace is a filter; however, the cc compiler is not, therefore you must send the output of ctrace to a temporary file before you can compile it with cc. In other words:

```
ctrace <myprog.c | cc
```

is not allowed. Instead you must use something like

```
ctrace <myprog.c >mytmp.c
cc mytmp.c
rm mytmp.c
```

ctrace allows a file argument to be specified, as in:

```
ctrace myprog.c >mytmp.c
```

ctrace detects loops and stops the trace until the loop is exited. A warning message is printed every 1000 times through the loop. The output of ctrace always goes to the standard output. A typical use would be to pipe the output of a crashing program to the UNIX tail command, and thus see the last few statements executed, as well as their effect.

ctrace has the following options:

-f *function_list*
-v *function list*
: Select only (-f) or all except (-v) the functions named in *function_list* for tracing.

-o -x -u -e
: Change the numeric formats from their defaults to octal (-o), hexadecimal -x) unsigned (-u), or floating point (-e).

-l *number*
: Specify the number of consecutive statements that are checked to detect loops (default 20). In other words, loops are assumed to be no more than *number* statements long. If you want every statement traced (that is, no loop checking), use the value 0.

-t *number*
: Specify the maximum number of variables that will be traced per statement. The default is 10, and the maximum is 20.

-P Run the preprocessor on the file before adding trace statements. This is useful if the file `#include`s some other file which must also be traced, or if you have defined macros which are expressions and you want them traced as well.

-p '*string*'
: Change the default (`'printf('`) trace statement. This option makes it possible to print the trace to other files. For example, to print all trace statements to stderr, one could use the option `-p 'fprintf(stderr,'`.

We can control the trace from within our program. There are two functions provided by `ctrace` to turn tracing off and on. These are `ctroff()` and `ctron()` respectively. `ctrace` also defines the symbol `CTRACE`, thus we can have conditionally compiled statements controlling the trace in an arbitrarily complex way. For example:

```
#ifdef CTRACE
   ctroff();
#endif
⋮
#ifdef CTRACE
   if( feof(outfile) && (n<100) ) ctron();
#endif
```

We can also use these functions from within sdb if we compile the output of ctrace with the -g option of cc. For example:

```
main:1b ctroff()
main:10b ctron()
r
```

in sdb will trace all but the first 10 lines of main.

Note that ctrace can't handle printing components of aggregate data types such as structures, unions and arrays.

C.8 cxref – The C Program Cross-Reference Generator

This utility analyses a collection of files and attempts to build a cross-reference for them. This cross-reference can contain all identifiers (including preprocessor names), either separately for each file, or in combination. The identifiers are listed together with all their occurrences. Defining references are marked with an asterisk *.

The syntax of the command is:

cxref [*options*] *files*

cxref allows the -D, -I, and -U options of cc, as well as:

-c	Produce a combined cross-reference for all files.
-o *file*	Send output to the named file.
-s	Don't print the input file names.
-t	Format output for 80-column device.

C.9 cb – The C Program Beautifier

cb is a filter which takes a C program (either from its argument list, or stdin), and writes it on stdout with appropriate spacing and indentation. Note that punctuation which is hidden in preprocessor statements may cause indentation errors. The available options are:

-s	Output code in Kernighan and Ritchie style.
-j	Rejoin split lines if necessary.
-l*length*	Split lines longer than *length*.

C.10 Building Your Own Libraries

The advantage of writing small, generalised functions is that they may be easily reused. If you write a program, and this program has, say, a few functions which are of general use and not specific to the program, you can separate these functions from the program and put them in your own library for future use.

A library (or *archive*) is a single file which contains other files. There are no restrictions on the contents of an archive; within a particular archive we may keep documents, program listings, compiled programs, and so on. Archive file names have the convention that they begin with **lib** and end with **.a**.

When you compile a program, you can specify the name of an archive or archives for ld to search. ld will extract any files from the archive that resolve external references in the program being compiled, and add them to the program. Note that library functions stored in an archive must have been compiled, but not linked. In other words, they must have been compiled with the -c option of cc.

The UNIX ar utility is used for archive maintenance. To create an archive (say libme.a) containing files fun1.o, fun2.o, and fun3.o, we use:

```
ar rv libme.a fun1.o fun2.o fun3.o
```

To extract fun1.o and fun3.o and place them in the current directory, we use:

```
ar xv libme.a fun1.o fun3.o
```

If we use the ar rv command shown above after having already created the archive, we will simply add the files fun1.o, fun2.o and fun3.o to the archive. If there are already files with that name in the archive, they will be replaced.

To examine the files in the archive, use:

```
ar tv libme.a
```

The only problem with these archives is that they are searched sequentially by ld, so functions which call other functions in the archive must precede those functions in the archive. This allows ld to detect an unresolved reference in a single pass, because when it reaches the called function it can resolve the reference. If the called function preceded the caller, and no other function called this function, then ld will skip the function since it does not resolve any references at that point. When ld eventually reaches the caller, an unresolved reference will occur that could not be resolved in the same pass. Fortunately there are solutions to this problem.

The first is for those UNIX systems that have the command ranlib. This command builds a table of references of a library, so that all the necessary functions can be quickly extracted. This is the ideal option, but not all UNIX systems have this command.

An alternative is to use the lorder and tsort commands. These produce a sorted list of function dependencies, which eliminates any forward references, except in the case when two functions refer to each other. To find the ordering for all the .o files in the current directory, we use:

```
lorder *.o | tsort
```

UNIX allows us to replace commands by their output by using backward quotes ‘ ‘. Thus we can create a sorted archive (called, say, libme) of all the .o files in the current directory, by using

```
ar -cr libme ‘lorder *.o | tsort‘
```

The last alternative, which may be necessary if we don't have ranlib, is for ld to search the library twice. This can be done by using the -lme option to ld twice (as well as -L*dir* to indicate which directory the library is stored in).

Note that UNIX System V supports random libraries automatically through ar.

C.11 Obtaining Information On Compiled Files

There are two commands that provide information on programs that have been compiled:

size prints the sizes of the sections (.text, .bss, .data, or .code) of files. The syntax is as follows:

size [*options*] *files*

The options are as follows:

-o	Select octal output.
-x	Select hexadecimal output.
-V	Print the version of size being used.

The sizes are printed in decimal by default.

nm prints selected symbol table information from compiled files. The information include symbol names, addresses, storage classes and sections of symbols. If the files were compiled with the -g option, the types, sizes and lines at which the symbols were declared are also printed. The format of the command is

nm [*options*] *files*

where the options are:

-o	Select octal output
-x	Select hexadecimal output
-h	Don't print a heading
-v	Sort the symbols by value
-n	Sort the symbols by name
-e	Select only the static and extern symbols
-u	Select only the undefined symbols
-f	Print all symbols in the symbol table
-V	Print the version of nm being used
-T	Truncate symbol names that are too long to be printed in the name column.

One other useful command is the time command.

time *command*

where *command* is an arbitrary command to the UNIX shell (and thus could be the execution of a program). This will print how long the command took to execute, as well as how much of this time was spent in the user part of the program and how much was spent in the system (for example, during I/O).

Appendix D

ANSI extensions to K&R C

This chapter considers some of the ANSI extensions to Kernighan and Ritchie C. These extensions are only supported by fairly recent compilers.

D.1 Preprocessor Enhancements

The preprocessor operator `##` allows concatenation of preprocessor arguments. Two simple examples should suffice as illustration. Given

```
#define JOIN(a,b)     a##b
#define PLURAL(a)     a##s
```

then

The call	**Expands to**
`JOIN(cat,mat)`	`catmat`
`PLURAL(cat)`	`cats`

The operator `#` converts a macro argument to a string. For example, given the definition

```
#define WRITE(s)     printf(#s)
```

then `WRITE(cat)` will expand to `printf("cat")`

The `#error` preprocessor command causes the preprocessor (and compiler) to print the specified message and stop compiling. For example,

```
#ifndef NULL
#error "stdio.h must be included"
#endif
```

There is also a `#pragma` preprocessor command used to pass information to the compiler, but that is beyond the scope of this book.

The ANSI standard includes several predefined macros:

`__DATE__`	the date of compilation
`__TIME__`	the time of compilation
`__FILE__`	the name of the source file being compiled
`__STDC__`	a nonzero constant
`__LINE__`	the current line number

The first four of these are strings; the last two are integers. The __STDC__ macro is used to indicate that the version of C being used complies with the ANSI standard; if this macro is not defined, then your C compiler does not comply with the standard.

D.2 Type Modifiers – `signed`, `const` and `volatile`

In addition to the `unsigned` type modifier, a `signed` modifier has been added. This was previously unnecessary, as the default was always to use signed representation for characters and integers. The use of `signed` allows this to be made explicit. Some compilers also have options allowing the user to specify whether the default `char` type should be signed or unsigned, and in the latter case, the `signed` modifier is needed to override the default.

ANSI C supports constants by using the type modifier `const`. In K&R C, constants were always handled by using preprocessor macros. The `const` type modifier is more flexible, allowing arbitrary type constants (for example, constant arrays can be declared). Obviously, a constant must be initialised when it is declared.

Pointer `const`s are also supported. In this case, the pointer points to a fixed address. The object at that address may be modified, but the pointer cannot be changed. For example:

```
const unsigned long *clock_tick = (unsigned long *)0x512;
```

declares a constant pointer which points to a `long` situated at address 512_{16} (which happens to be the clock tick counter on an IBM PC). The following would then be legal under MS-DOS[1]:

```
*clock_tick = 0l;     /* Reset the tick counter */
```

[1]MS-DOS is a trademark of Microsoft Corporation.

since it is assigning to a variable which is pointed to by a constant. However, the following is **not** legal:

```
clock_tick = NULL;
```

as it is attempting to assign to a constant.

The type modifier `volatile` tells the compiler not to perform any optimisations on the specified variables. The reasons for using this modifier are beyond the scope of this book.

D.3 Function Prototypes

A characteristic of K&R C is the poor type checking that is performed by the compiler. In other words, the compiler will typically not report any problems with the following:

```
void funk(a,b)
   char a;
   int b;
{
   ⋮
}
long x, y;
⋮
funk(x,y);
⋮
```

This is obviously incorrect, as we are passing a total of eight bytes (two `long`s) as parameters to a function which expects four bytes (a `char`, which is cast to `int`, and an `int`. It is this vulnerability to errors which resulted in the development of C program checkers such as lint.

ANSI C supports a different syntax for function declarations which allows strong type checking and avoids these types of errors. Pascal programmers will find the new syntax, called *function prototyping*, much more familiar. To declare a function using prototypes, the parameter types are specified within the argument list itself. The function must be declared before any in order calls for the type checking to be used. This may require forward declarations in some cases. Often, each global function has a prototype declaration in a header file, which is included by files which call the functions. For example, compilers that support ANSI C will use function prototypes for the input and output routines, and declare these prototypes in the **stdio.h** file. Thus, by including **stdio.h**, all calls to standard I/O functions have strong type checking applied.

The use of prototypes is best illustrated by an example. Consider the function declaration:

```
void funk(a,b,c)
   int a;
   char *c;
   float b[];
{
   ⋮
}
```

In K&R C, a forward declaration of such a function would look like

```
void funk(a,b,c);
```

Notice that this gives no information about the parameter types. Using prototypes, the function would be declared as:

```
void funk(int a, float b[], char *c)
{
   ⋮
}
```

and a forward declaration would be

```
void funk(int a, float b[], char *c);
```

giving complete type information on the function type as well as the parameters. Our original erroneous example would immediately be reported by the compiler if we had used prototypes. In fact, prototypes eliminate much of the need for checkers such as lint.

Appendix E

C Style and Coding Standards

Recommended C Style and Coding Standards

L.W. Cannon, R.A. Elliott
L.W. Kirchhoff, J.H. Miller
J.M. Milner, R.W. Mitze
E.P. Schan, N.O. Whittington
Bell Labs

Henry Spencer
Zoology Computer Systems
University of Toronto

David Keppel
EECS, UC Berkeley
CS, University of Washington

November 18, 1989

Abstract

This document is an updated version of the Indian Hill C Style and Coding Standards paper, with modifications by the last two authors. It describes a recommended coding standard for C programs. The scope is coding style, not functional organization.

E.1 Introduction

This document is a modified version of a document from a committee formed at Indian Hill to establish a common set of coding standards and recommendations for the Indian Hill community. The scope of this work is C coding style, rather than the functional organization of programs or general issues such as the use of gotos. We[1] have tried to combine previous work [1,6,8] on C style into a uniform set of standards that should be appropriate for any project using C, although parts are biased towards particular systems. Of necessity, these standards cannot cover all situations. Experience and informed judgement count for much. Programmers who encounter unusual situations should consult (1) experienced C programmers or (2) code written by experienced C programmers, preferably following these rules.

The standards in this document are not of themselves required, but individual institutions or groups may adopt part or all of them as a part of program acceptance. It is therefore likely that others at your institution will code in a similar style. Ultimately, the goal of these standards is to increase portability, reduce maintenance, and above all improve clarity.

Many of the style choices here are somewhat arbitrary. Mixed coding style is harder to maintain than bad coding style. When changing existing code it is better to conform to the style (indentation, spacing, commenting, naming conventions) of the existing code than it is to blindly follow this document.

"To be clear is professional; not to be clear is unprofessional." — Sir Ernest Gowers.

E.2 File Organization

A file consists of various sections that should be separated by several blank lines. Although there is no maximum length limit for source files, files with more than about 1000 lines are cumbersome to deal with. The editor may not have enough temp space to edit the file, compilations will go more slowly, etc. Many rows of asterisks, for example, present little information compared to the time it takes to scroll past, and are discouraged. Lines longer than 80 columns are not handled well by all terminals and should be

[1]The opinions in this document do not reflect the opinions of all authors. This is still an evolving document. Please send comments and suggestions to pardo@cs.washington.edu or (rutgers,cornell,ucsd,ubc-cs,tektronix)!uw-beaver!june!pardo

avoided if possible. Excessively long lines which result from deep indenting are often a symptom of poorly-organized code.

E.2.1 File Naming Conventions

File names are made up of a base name, and an optional period and suffix. The first character of the name should be a letter and all characters (except the period) should be all lowercase letters and numbers. The base name should be 8 or fewer characters and the suffix should be 3 or fewer characters (four, if you include the period). These rules apply to both program files and default files used and produced by the program (e.g., "rogue.sav").

Some compilers and tools require and/or use certain suffix conventions for names of files [5]. The following suffixes are required:

- C source file names must end in *.c*
- Assembler source file names must end in *.s*

The following conventions are universally followed:

- Relocatable object file names end in *.o*
- Include header file names end in *.h* [2].
- Yacc source file names end in *.y*
- Lex source file names end in *.l*

C++ has compiler-dependent suffix conventions, including *.c*, *..c*, *.cc*, *.c.c*, and *.C*. Since much C code is also C++ code, there is no clear solution.

In addition, it is conventional to use 'Makefile' (not 'makefile') for the control file for *make* (for systems that support it) and 'README' for a summary of the contents of the directory or directory tree.

E.2.2 Program Files

The suggested order of sections for a program file is as follows:

[2] An alternate convention that may be preferable in multi-language environments is to suffix both the language type and *.h* (e.g. "foo.c.h" or "foo.ch").

1. First in the file is a prologue that tells what is in that file. A description of the purpose of the objects in the files (whether they be functions, external data declarations or definitions, or something else) is more useful than a list of the object names. The prologue may optionally contain author(s), revision control information, references, etc.

2. Any header file includes should be next. If the include is for a non-obvious reason, the reason should be commented. In most cases, system include files like *stdio.h* should be included before user include files.

3. Any defines and typedefs that apply to the file as a whole are next. One normal order is to have "constant" macros first, then "function" macros, then typedefs and enums.

4. Next come the global (external) data declarations, usually in the order: externs, non-static globals, static globals. If a set of defines applies to a particular piece of global data (such as a flags word), the defines should be immediately after the data declaration or embedded in structure declarations, indented to put the *defines* one level deeper than the first keyword of the declaration to which they apply.

5. The functions come last, and should be in some sort of meaningful order. Like functions should appear together. A "breadth-first" approach (functions on a similar level of abstraction together) is preferred over depth-first (functions defined as soon as possible before or after their calls). Considerable judgement is called for here. If defining large numbers of essentially-independent utility functions, consider alphabetical order.

E.2.3 Header Files

Header files are files that are included in other files prior to compilation by the C preprocessor. Some are defined at the system level like *stdio.h* which must be included by any program using the standard I/O library. Header files are also used to contain data declarations and defines that are needed by more than one program. Header files should be functionally organized, i.e., declarations for separate subsystems should be in separate header files. Also, if a set of declarations is likely to change when code is ported from one machine to another, those declarations should be in a separate header file.

Avoid private header filenames that are the same as library header filenames. The statement `#include "math.h"` will include the standard library math header file if the intended one is not found in the current directory. If this is what you want to happen, comment this fact. Don't use absolute pathnames for header files. Use the <name> construction for getting them from a standard place, or define them relative to the current directory. The "include-path" option of the C compiler (-I on many systems) is the best way to handle extensive private libraries of header files; it permits reorganizing the directory structure without having to alter source files.

Defining variables in a header file is often a poor idea. Frequently it is a symptom of poor partitioning of code between files. Some objects like typedefs and initialized data definitions cannot be seen twice by the compiler in one compilation. On some systems, repeating uninitialized declarations without the *extern* keyword also causes problems. Repeated declarations can happen if include files are nested and will cause the compilation to fail.

Header files should not be nested. The prologue for a header file should, therefore, describe what other headers need to be #included for the header to be functional. In extreme cases, where a large number of header files are to be included in several different source files, it is acceptable to put all common #includes in one include file.

It is common to put the following into each .h file to prevent accidental double-inclusion.

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

... /* body of example.h file */

#endif /* EXAMPLE_H */
```

This double-inclusion mechanism should not be relied upon, particularly to perform nested includes.

E.3 Comments

"When the code and the comments disagree, both are probably wrong." — *Norm Schreyer*

The comments should describe what is happening, how it is being done, what parameters mean, which globals are used and which are modified, and any restrictions or bugs. Avoid, however, comments that are clear from the code. Such information rapidly gets out of date. Comments that disagree with the code are of negative value. Short comments should be *what* comments, such as "compute mean value", rather than *how* comments such as "sum of values divided by n". C is not assembler; putting a comment at the top of a 3-10 line section telling what it does overall is often more useful than a comment on each line describing micrologic.

Comments should justify offensive code. The justification should be that something bad will happen if unoffensive code is used. Just making code faster is not enough to rationalize a hack; the performance must be *shown* to be unacceptable without the hack. The comment should explain the unacceptable behavior and describe why the hack is a "good" fix.

Comments that describe data structures, algorithms, etc., should be in block comment form with the opening /* in column one, a * in column 2 before each line of comment text, and the closing */ in columns 2-3. An alternative is to have ** in column 1-2, and put the closing */ also in 1-2.

```
/*
 *      Here is a block comment.
 *      The comment text should be tabbed or spaced over uniformly.
 *      The opening slash-star and closing star-slash are alone on a line.
 */

/*
** Alternate format for block comments
*/
```

Note that *grep* * will catch all block comments in the file [3]. Very long block comments such as drawn-out discussions and copyright notices often start with /* in column one, no leading * before lines of text, and the closing */ in columns 1-2. Block comments inside a function are appropriate, and they should be tabbed over to the same tab setting as the code that they describe. One-line comments alone on a line should be indented to the tab setting of the code that follows.

```
 if (argc > 1) {
```

[3]Some automated program-analysis packages use different characters before comment lines as a marker for lines with specific items of information. In particular, a line with a '-' in a comment preceding a function is sometimes assumed to be a one-line summary of the function's purpose.

```
        /* Get input file from command line. */
        if (freopen(argv[1], "r", stdin) == NULL) {
                perror (argv[1]);
        }
}
```

Very short comments may appear on the same line as the code they describe, and should be tabbed over to separate them from the statements. If more than one short comment appears in a block of code they should all be tabbed to the same tab setting.

```
if (a == 2) {
        return(TRUE);                   /* special case */
}  else  {
        return(isprime(a));             /* works only for odd a */
}
```

E.4 Declarations

Global declarations should begin in column 1. All external data declaration should be preceded by the extern keyword. If an external variable is an array that is defined with an explicit size, then the array bounds must be repeated in the extern declaration unless the size is always encoded in the array (e.g., a read-only character array that is always null-terminated). Repeated size declarations are particularly beneficial to someone picking up code written by another.

The “pointer” qualifier, ‘*’, should be with the variable name rather than with the type.

```
char     *s, *t, *u;
```

instead of

```
char*    s, t, u;
```

Unrelated declarations, even of the same type, should be on separate lines. A comment describing the role of the object being declared should be included, with the exception that a list of #defined constants do not need comments if the constant names are sufficient documentation. The names, values, and comments should be tabbed so that they line up underneath each other. Use the tab character rather than blanks. For structure and union template declarations, each element should be alone on a line with a comment describing it. The opening brace ({) should be on the same line as the structure tag, and the closing brace (}) should be in column 1.

```
struct boat {
       int     wllength;       /* water line length in meters */
       int     type;           /* see below */
       long    sailarea;       /* sail area in square mm */
};

/*
 * defines for boat.type
 */
#              define KETCH (1)
#              define YAWL  (2)
#              define SLOOP (3)
#              define SQRIG (4)
#              define MOTOR (5)
```

These defines are sometimes put right after the declaration of type, within the struct declaration, with enough tabs after the '#' to indent define one level more than the structure member declarations. When the actual values are unimportant, the enum facility is better [4].

```
enum bt_t { KETCH, YAWL, SLOOP, SQRIG, MOTOR };
struct boat {
       int             wllength;       /* water line length in meters */
       enum bt_t       type;           /* what kind of boat */
       long            sailarea;       /* sail area in square mm */
};
```

Any variable whose initial value is important should be explicitly initialized, or at the very least should be commented to indicate that C's default initialization to zero is being relied upon. The empty initializer, "{}", should never be used. Structure initializations should be fully parenthesized with braces. Constants used to initialize longs should be explicitly long.

[4]enums might be better anyway

```
int            x = 1;
char           *msg = "message";
struct boat    winner[] = {
       { 40, YAWL, 6000000L },
       { 28, MOTOR, 0L },
       { 0 },
};
```

In any file which is part of a larger whole rather than a self-contained program, maximum use should be made of the static keyword to make functions and variables local to single files. Variables in particular should be accessible from other files only when there is a clear need that cannot be filled in another way. Such usages should be commented to make it clear that another file's variables are being used; the comment should name the other file. If your debugger hides static objects you need to see during debugging, declare them as STATIC and #define STATIC as needed.

The most important few types should be highlighted by typedeffing them, even if they are only integers, as the unique name makes the program easier to read (as long as there are only a *few* things typedeffed to integers!). Structures may be typedeffed when they are declared. Give the struct and the typedef the same name.

```
typedef struct splodge_t {
        int sp_count;
        char *sp_name, *sp_alias;
} splodge_t;
```

The return type of functions should always be declared. If function prototypes are available, use them. One common mistake is to omit the declaration of external math functions that return double. The compiler then assumes that the return value is an integer and the bits are dutifully converted into a (meaningless) floating point value.

E.5 Function Declarations

Each function should be preceded by a block comment prologue that gives a short description of what the function does and (if not clear) how to use it. Discussion of non-trivial design decisions and side-effects is also appropriate. Avoid duplicating information clear from the code.

The function return value should be alone on a line, indented one stop[5]. Do not default to int; if the function does not return a value then it should be given return type void[6]. If the value returned requires a long explanation, it should be given in the prologue; otherwise it can be on the same line as the return type, tabbed over. The function name (and the formal parameter list) should be alone on a line, in column 1. Destination (return value) parameters should generally be first (on the left). All formal parameter declarations, local declarations and code within the function body should be tabbed over one stop. The opening brace of the function body should be alone on a line beginning in column 1.

Each parameter should be declared (do not default to int). In general each variable declaration should be on a separate line with a comment describing the role played by the variable in the function. Loop counters called "i", and string pointers called "s" are typically excluded. If a group of functions all have a like parameter or local variable, it helps to call the repeated variable by the same name in all functions. Like parameters should also appear in the same place in the various argument lists.

Comments for parameters and local variables should be tabbed so that they line up underneath each other. Local variable declarations should be separated from the function's statements by a blank line.

Be careful when you use or declare functions that take a variable number of arguments ("varargs"). There is no truly portable way to do varargs in C. Better to design an interface that uses a fixed number of arguments. If you must have varargs, use the library macros for declaring functions with variant argument lists.

If the function uses any external variables (or functions) that are not declared globally in the file, these should have their own declarations in the function body using the *extern* keyword.

Avoid local declarations that override declarations at higher levels. In particular, local variables should not be redeclared in nested blocks. Although this is valid C, the potential confusion is enough that *lint* will complain about it when given the -h option.

[5] "Tabstops" can be blanks (spaces) inserted by your editor in clumps of 2, 4, or 8. Use actual tabs where possible.

[6] #define void or #define void int for compilers without the void keyword.

E.6 Whitespace

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Dishonorable mention, Obfuscated C Code Contest, 1984. Author requested anonymity.

Use whitespace generously, vertically and horizontally. Indentation and spacing should reflect the block structure of the code; e.g., there should be at least 2 blank lines between the end of one function and the comments for the next.

A long string of conditional operators should be split onto separate lines.

```
if (foo->next==NULL && totalcount<needed && needed<=MAX_ALLOT
       && server_active(current_input)) { ...
```

might be better as

```
if (foo->next == NULL
       && totalcount < needed
       && needed <= MAX_ALLOT
       && server_active(current_input))  {   ...
```

Similarly, elaborate for loops should be split onto different lines.

```
for (curr = *listp, trail = listp;
       curr != NULL;
       trail = &(curr->next), curr = curr->next )
{
       ...
```

Other complex expressions, particularly those using the ternary (?:) operator, are best split on to several lines, too.

```
c = (a == b)
       ? d + f(a)
       : f(b) - d;
```

Examples

```
/*
 *     Determine if the sky is blue by checking that it isn’t night.
 *     CAVEAT: Only sometimes right. May return TRUE when the answer
 *     is FALSE.
 *     NOTE: Uses ‘hour’ from ‘hightime.c’. Returns ‘int’ for
 *     compatibility with the old version.
 */
       int                            /* TRUE or FALSE */
 skyblue()
 {
       extern int      hour;          /* current hour of the day */

       if (hour < MORNING || hour > EVENING) {
               return (FALSE);        /* black */
       } else {
               return (TRUE);         /* blue */
       }
 }



/*
 *      Find the last element in the linked list
 *      pointed to by nodep and return a pointer to it.
 *      Return NULL if there is no last element.
 */
        node_t *
tail(nodep)
        node_t          *nodep;       /* pointer to head of list */
{
        register node_t *np;          /* advances to NULL */
        register node_t *lp;          /* follows one behind np */

        if (nodep == NULL)
                return (NULL);
        np = lp = nodep;
        while ((np = np->next) != NULL) {
                lp = np;
        }
        return (lp);
}
```

E.7 Simple Statements

There should be only one statement per line unless the statements are very closely related.

```
case FOO:   oogle (zork);  boogle (zork);  break;
case BAR:   oogle (bork);  boogle (zork);  break;
case BAZ:   oogle (gork);  boogle (bork);  break;
```

Always document a null body for a for or while statement so that it is clear that the null body is intentional and not missing code.

```
while (*dest++ = *src++)
        ;         /* VOID */
```

Do not default the test for non-zero, i.e.

```
if (f() != FAIL)
```

is better than

```
if (f())
```

even though FAIL may have the value 0 which C considers to be false. An explicit test will help you out later when somebody decides that a failure return should be -1 instead of 0. Explicit comparison should be used even if the comparison value will never change; e.g., “if (!(bufsize % sizeof(int)))” should be written instead as “if ((bufsize % sizeof(int)) == 0)” to reflect the numeric (not boolean) nature of the test. A frequent trouble spot is using strcmp to test for string equality, where the result should *never ever* be defaulted. The preferred approach is to define a macro *STREQ*.

```
#define STREQ(a, b) (strcmp((a), (b)) == 0)
```

The non-zero test is often defaulted for predicates and other functions or expressions which meet the following restrictions:

- Returns 0 for false, nothing else.

- Is named so that the meaning of (say) a 'true' return is absolutely obvious. Call a predicate isvalid or valid, not checkvalid.

It is common practice to declare a boolean type "bool" in a global include file. The special names improve readability immensely.

```
typedef int     bool;
#define FALSE   0
#define TRUE    1
```

or

```
typedef enum { NO=0, YES } bool;
```

Even with these declarations, do not check a boolean value for equality with 1 (TRUE, YES, etc.); instead test for inequality with 0 (FALSE, NO, etc.). Most functions are guaranteed to return 0 if false, but only non-zero if true. Thus,

```
if (func() == TRUE) { ...
```

must be written

```
if (func() != FALSE) { ...
```

There is a time and a place for embedded assignment statements. In some constructs there is no better way to accomplish the results without making the code bulkier and less readable.

```
while ((c = getchar()) != EOF) {
        process the character
}
```

The `++` and `--` operators count as assignment statements. So, for many purposes, do functions with side effects. Using embedded assignment statements to improve run-time performance is also possible. However, one should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,

```
a = b + c;
d = a + r;
```

should not be replaced by

```
 d = (a = b + c) + r;
```

even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

Goto statements should be used sparingly, as in any well-structured code. The main place where they can be usefully employed is to break out of several levels of switch, for, and while nesting, although the need to do such a thing may indicate that the inner constructs should be broken out into a separate function, with a success/failure return code.

```
        for (...) {
                while (...) {
                        ...
                        if (disaster)
                                goto error;

                }
        }
        ...
error:
        clean up the mess
```

When a goto is necessary the accompanying label should be alone on a line and tabbed one stop to the left of the code that follows. The goto should be commented (possibly in the block header) as to its utility and purpose. Continue should be used sparingly and near the top of the loop. Break is less troublesome.

E.8 Compound Statements

A compound statement is a list of statements enclosed by braces. There are many common ways of formatting the braces. Be consistent with your local standard, if you have one, or pick one and use it consistently. When editing someone else's code, *always* use the style used in that code.

```
control {
       statement;
       statement;
}
```

The style above is called "K&R style", and is preferred if you haven't already got a favorite. With K&R style, the *else* part of an *if-else* statement and the *while* part of a *do-while* statement should appear on the same line as the close brace. With most other styles, the braces are always alone on a line.

When a block of code has several labels (unless there are a lot of them), the labels are placed on separate lines. The fall-through feature of the C *switch* statement, (that is, when there is no break between a code segment and the next case statement) must be commented for future maintenance. A lint-style comment/directive is best.

```
switch (expr) {
        case ABC:
        case DEF:
                statement;
                break;
        case UVW:
                statement;
                /*FALLTHROUGH*/
        case XYZ:
                statement;
                break;
}
```

Here, the last break is unnecessary, but is required because it prevents a fall-through error if another case is added later after the last one. The default case, if used, should be last and does not require a break.

Whenever an if-else statement has more than one statement in the if or else section, the statements of both the if and else sections should both be enclosed in braces (called *fully bracketed syntax*).

```
if (expr) {
        statement;
} else {
        statement;
        statement;
}
```

An *if-else* with many *else if* statements should be written with the *else* conditions left-justified.

```
if (STREQ (reply, "yes")) {
        statements for yes
        ...
} else if (STREQ (reply, "no")) {
        ...
} else if (STREQ (reply, "maybe")) {
        ...
} else {
        statements for default
        ...
}
```

The format then looks like a generalized *switch* statement and the tabbing reflects the switch between exactly one of several alternatives rather than a nesting of statements.

The following code is very dangerous:

```
#ifdef CIRCUIT
#       define CLOSE_CIRCUIT(circno)     { close_circ(circno); }
#else
#       define CLOSE_CIRCUIT(circno)
#endif


        ...
        if (expr)
                statement;
        else
                CLOSE_CIRCUIT(x)
        ++i;
```

Note that on systems where CIRCUIT is not defined the statement "++i;" will only get executed when expr is false! This example points out both the value of naming macros with CAPS and of making code fully-bracketed.

E.9 Operators

Generally, all binary operators except '.' and '->' should be separated from their operands by blanks. Some judgement is called for in the case of complex expressions, which may be clearer if the "inner" operators are not surrounded by spaces and the "outer" ones are. In addition, keywords that are followed by expressions in parentheses should be separated from the left parenthesis by a blank. (Sizeof is an exception.) Blanks should also appear after commas in argument lists to help separate the arguments visually. On the other hand,macro definitions with arguments must not have a blank between the name and the left parenthesis. The C preprocessor requires the left parenthesis to be immediately after the macro name or else the argument list will not be recognized. Unary operators should not be separated from their single operand.

If you think an expression will be hard to read, consider breaking it across lines. Splitting at the lowest-precedence operator near the break is best. Since C has some unexpected precedence rules, expressions involving mixed operators should be parenthesized. Too many parenthesis, however, can make a line *harder* to read because humans aren't good at parenthesis-matching.

There is a time and place for the binary comma operator, but generally it should be avoided. The comma operator is most useful to provide multiple initializations or operations, as in *for* statements. Complex expressions, for instance those with nested ?: (ternary) operators, can be confusing and should be avoided if possible. There are some macros like getchar where both the ternary operator and comma operators are useful. The logical expression operand before the ?: should be parenthesized and both return values must be the same type.

E.10 Naming Conventions

Individual projects will no doubt have their own naming conventions. There are some general rules however.

- Names with leading and trailing underscores are reserved for system purposes and should not be used for any user-created names. Most systems use them for names that the user should not have to know. If you must have your own private identifiers, begin them with a letter or two identifying the package to which they belong.

- #define constants should be in all CAPS.
- Enum tags are Capitalized or in all CAPS
- Function, structure tag, typedef, and variable names should be in lower case.
- Many macro "functions" are in all CAPS. Some macros (such as getchar and putchar) are in lower case since they may also exist as functions. Lower-case macro names are only acceptable if the macros behave like a function call, that is, they evaluate their parameters exactly once and do not assign values to named parameters. Sometimes it is impossible to write a macro that behaves like a function even though the arguments are evaluated exactly once.
- Avoid names that differ only in case, like foo and Foo. Similarly, avoid foobar and foo_bar. The potential for confusion is considerable.

In general, global names (including enums) should have a common prefix identifying the module that they belong with. They may alternatively be grouped in a global structure. Typedeffed names often have "_t" appended to their name.

Avoid names that might conflict with various standard library names. Some systems will include more library code than you want. Also, your program may be extended someday.

E.11 Constants

Numerical constants should not be coded directly. Symbolic constants make the code easier to change and easier to read. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value.

The #define feature of the C preprocessor should be used to give constants meaningful names. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the #define. The enumeration data type is a better way to declare variables that take on only a discrete set of values, since additional type checking is often available.

Constants should be defined consistently with their use; e.g. use 540.0 for a float instead of 540 with an implicit float cast. There are some cases where the constants 0 and 1 may appear as themselves instead of as defines. For example if a for loop indexes through an array, then

```
for (i = 0; i < ARYBOUND; i++)
```

is reasonable while the code

```
qval = opens(door[i], 7);
if (qval == 0)
        error("can't open %s\n", door[i]);
```

is not. In the last example qval is a pointer. When a value is a pointer it should be compared to NULL instead of 0. NULL is available either as part of the standard I/O library's header file *stdio.h* or in *stdlib.h* for newer systems. Even simple values like 1 or 0 are often better expressed using defines like TRUE and FALSE (sometimes YES and NO read better).

Simple character constants should be defined as character literals rather than numbers. Non-text characters are discouraged as non-portable. If non-text characters are necessary, particularly if they are used in strings, they should be written using a escape character of three octal digits rather than one (e.g. '\007'). Such usage should be considered machine-dependent and treated as such.

E.12 Macros

Complex expressions can be used as macro parameters, and operator-precedence problems can arise unless all occurrences of parameters have parentheses around them. There is little that can be done about the problems caused by side effects in parameters except to avoid side effects in expressions (a good idea anyway) and, when possible, to write macros that evaluate their parameters exactly once. There are times when it is impossible to write macros that act exactly like functions.

Some macros also exist as functions (e.g., getc and fgetc). The macro should be used in implementing the function so that changes to the macro will be automatically reflected in the function. Care is needed when interchanging macros and functions since functions pass their parameters by value whereas macros pass their arguments by name substitution. Carefree use of macros requires care when they are defined.

Macros should avoid using globals, since the global name may be covered by a local declaration. Macros that change named parameters (rather than the storage they point at) or may be used as the left-hand side of an assignment should mention this in their comments. Macros that take no parameters but reference variables, are long, or are aliases for function calls should be given an empty parameter list, e.g.,

```
#define OFF_A() (a_global+OFFSET)
#define BORK() (zork())
#define SP3() if (b) { av+=1; bv+=1; cv+=1; }
```

Macros save function call/return overhead, but when a macro gets long, the effect of the call/return becomes negligible, so a function should be used instead.

In some cases it is appropriate to make the compiler insure that a macro is terminated with a semicolon.

```
if (x==3)
       SP3();
else
       BORK();
```

If the semicolon is omitted after the call to SP3, then the else will (silently!) become associated with the if in the SP3 macro. With the semicolon, the else doesn't match any if! The macro SP3 can be written safely as

```
#define SP3() do { av+=1; bv+=1; cv+=1; } while (0)
```

Writing out the enclosing do-while by hand is awkward and some compilers and tools may complain that there is a constant in the "while" conditional. A macro for declaring statements may make programming easier.

```
#ifdef lint
       static int ZERO;
#else
#      define ZERO 0
#endif
#define STMT(stuff )       do { stuff } while (ZERO)
```

Declare SP3 with

```
#define SP3()       STMT( if (bool) { av+=1; bv+=1; cv+=1; } )
```

Using STMT will help prevent small typos from silently changing programs.

Except for hacks such as the above, macros should contain keywords only if the entire macro is surrounded by braces.

E.13 Debugging

If you use enums, the first tag should have a non-zero value, or the first tag should indicate an error.

```
enum { STATE_ERR, STATE_START, STATE_NORMAL, STATE_END } state_t;
enum {VAL_NEW=1, VAL_NORMAL, VAL_DYING, VAL_DEAD } value_t;
```

Uninitalized values will then often “catch themselves”.

Check for error return values, even from functions that “can’t” fail. Consider that close() and fclose() can and do fail, even when all prior file operations have succeeded. Write your own functions so that they test for errors and return error values or abort the program in a well-defined way. Include a lot of debugging and error-checking code and leave most of it in the finished product. Check even for “impossible” errors. [8]

Use the assert facility to insist that each function is being passed well-defined values, and that intermediate results are well-formed.

Build in the debug code using as few #ifdefs as possible. For instance, if “mm_malloc” is a debugging memory allocator, then MALLOC will select the appropriate allocator, avoids littering the code with #ifdefs, and makes clear the difference between allocation calls being debugged and extra memory that is allocated only during debugging.

```
#ifdef DEBUG
#       define MALLOC(size) (mm_malloc(size))
#else
#       define MALLOC(size) (malloc(size))
#endif
```

Check bounds even on things that “can’t” overflow. A function that writes on to variable-sized storage should take an argument maxsize that is the size of the destination. If there are times when the size of the destination is unknown, some ‘magic’ value of maxsize should mean “no bounds checks”. When bound checks fail, make sure that the function does something useful such as abort or return an error status.

```
/*
 *      INPUT: A null-terminated source string ‘src’ to copy from and
 *      a ‘dest’ string to copy to. ‘maxsize’ is the size of ‘dest’
 *      or UINT_MAX if the size is not known. ‘src’ and ‘dest’ must
 *      both be shorter than UINT_MAX, and ‘src’ must be no longer than
 *      ‘dest’.
 *      OUTPUT: The address of ‘dest’ or NULL if the copy fails.
 *      ‘dest’ is modified even when the copy fails.
 */

        char *
copy (dest, maxsize, src)
        char *dest, *src;
        unsigned maxsize;
{
        char *retval = dest;

        while (*dest++ = *++src && maxsize-- > 0)
                ;               /* VOID */

        if (maxsize == 0)
                retval = NULL;

        return (retval);
}
```

In all, remember that a program that produces wrong answers twice as fast is infinitely slower. The same is true of programs that crash occasionally or clobber valid data.

“C Code. C code run. Run, code, run... PLEASE!!!” — Barbara Toungue

E.14 Conditional Compilation

Conditional compilation is useful for things like machine-dependencies, debugging, and for setting certain options at compile-time. Beware of conditional compilation. Various controls can easily combine in unforseen ways. If you #ifdef machine dependencies, make sure that when no machine is specified, the result is an error, not a default. If you #ifdef optimizations, the default should be the unoptimized code rather than an uncompilable program. Be sure to test the unoptimized code.

Put #ifdefs in header files instead of source files when possible. Use the #ifdefs to define macros that can be used uniformly in the code. For instance, a header file for checking memory allocation might look like (omitting definitions for REALLOC and FREE):

```
#ifdef DEBUG
       extern char *mm_malloc();
#      define MALLOC(size) (mm_malloc(size))
#else
       extern char *malloc();
#      define MALLOC(size) (malloc(size))
#endif
```

Conditional compilation should generally be on a feature-by-feature basis. Machine or operating system dependencies should be avoided in most cases.

```
#ifdef BSD4
       long t = time(((long *)NULL);
#endif
```

The preceding code is poor for two reasons: there may be 4BSD systems for which there is a better choice, and there may be non-4BSD systems for which the above is the best code. Instead, use define symbols such as TIME_LONG and TIME_STRUCT and define the appropriate one in a configuration file such as config.h.

E.15 Portability

"C combines the power of assembler with the portability of assembler." — Bill Thacker, misquoted by anonymous.

The advantages of portable code are well known. This section gives some guidelines for writing portable code. Here, "portable" means that a source file can be compiled and executed on different machines with the only change being the inclusion of possibly different header files and the use of different compiler flags. The header files will contain #defines and typedefs that may vary from machine to machine. In general, a new "machine" is different hardware, a different operating system, a different compiler, or any combination of these. Reference [1] contains useful information on both style and portability. The following is a list of pitfalls to be avoided and recommendations to be considered when designing portable code:

- Write portable code first, worry about detail optimizations only on machines where they prove necessary. Optimized code is often obscure. Optimizations for one machine may produce worse code on another. Document performance hacks and localize them as much as possible. Documentation should explain how it works and why it was needed (e.g., “loop executes 6 zillion times”).

- Recognize that some things are inherently non-portable. Examples are code to deal with particular hardware registers such as the program status word, and code that is designed to support a particular piece of hardware, such as an assembler or I/O driver. Even in these cases there are many routines and data organizations that can be made machine independent.

- Organize source files so that the machine-independent code and the machine-dependent code are in separate files. Then if the program is to be moved to a new machine, it is a much easier task to determine what needs to be changed. Comment the machine dependence in the headers of the appropriate files.

- Any behavior that is described as “implementation defined” should be treated as a machine (compiler) dependency. Assume that the compiler or hardware does it some completely screwy way.

- Pay attention to word sizes. Objects may be non-intuitive sizes, Pointers are not always the same size as ints, the same size as each other, or freely interconvertible. The following table shows bit sizes for basic types in C for various machines and compilers.

type	pdp11 series	vax	68000 family	Cray-2	Unisys 1100	Harris H800	80386
char	8	8	8	8	9	8	8
short	16	16	8/16	64(32)	18	24	8/16
int	16	32	16/32	64(32)	36	24	16/32
long	32	32	32	64	36	48	32
char*	16	32	32	64	72	24	16/32/48
int*	16	32	32	64(24)	72	24	16/32/48
int(*)	16	32	32	64	576	24	16/32/48

Some machines have more than one possible size for a given type. The size you get can depend both on the compiler and on various compile-time flags. The following table shows "safe" type sizes on the majority of systems. Unsigned numbers are the same bit size as signed numbers.

Type	Minimum # Bits	No Smaller Than
char	8	
short	16	char
int	16	short
long	32	int
float	24	
double	38	float
any *	14	
char *	15	any *
void *	15	any *

- The void* type is guaranteed to have enough bits of precision to hold a pointer to any data object. The void(*)() type is guaranteed to be able to hold a pointer to any function. Use these types when you need a generic pointer. (Use char* and char(*)(), respectively, in older compilers). Be sure to cast pointers back to the correct type before using them.

- Even when, say, a void* and a char* are the same *size*, they may have different *formats*. For example, the following will fail on some machines that have sizeof(int*) equal to sizeof(char*). The code fails because free expects a char* and gets passed an int*.

  ```
  int *p = (int *) malloc (sizeof(int));
  free (p);
  ```

- Note that the *size* of an object does not guarantee the *precision* of that object. The Cray-2 may use 64 bits to store an int, but a *long* cast into an int and back to a long may be truncated to 32 bits.

- The integer constant zero may be cast to any pointer type. The resulting pointer is called a *null pointer* for that type, and is different from any other pointer of that type. A null pointer always compares equal to the constant zero. A null pointer might *not* compare equal with a variable that has the value zero. Null pointers are *not* always stored with all bits zero. Null pointers for two different types are sometimes different. A null pointer of one type cast in to a pointer of another type will be cast in to the null pointer for that second type.

- On ANSI compilers, when two pointers of the same type access the same storage, they will compare as equal. When non-zero integer constants are cast to pointer types, they may become identical to other pointers. On non-ANSI compilers, pointers that access the same storage may compare as different. The following two pointers, for instance, may or may not compare equal, and they may or may not access the same storage.

  ```
  ((int *) 2 )
  ((int *) 3 )
  ```

  If you need 'magic' pointers other than NULL, either allocate some storage or treat the pointer as a machine dependence.

  ```
  extern int x_int_dummy;             /* in x.c */
  #define X_FAIL (NULL)
  #define X_BUSY (&x_int_dummy)

  #define X_FAIL (NULL)
  #define X_BUSY MD_PTR1              /* MD_PTR1 from "machine.h" */
  ```

- Floating-point numbers have both a *precision* and a *range*. These are independent of the size of the object. Thus, overflow (underflow) for a 32-bit floating-point number will happen at different values on different machines. Also, 4.99999999999 times 5.00000000001 will yield two different numbers on two different machines. Differences in rounding and truncation can give surprisingly different answers.

- On some machines, a double may have *less* range or precision than a float.

- On some machines the first half of a double may be a float with similar value. Do *not* depend on this.

- Watch out for signed characters. On the VAX, for instance, characters are sign extended when used in expressions, which is not the case on many other machines. Code that assumes signed/unsigned is unportable. For example, a[c] won't work if c is supposed to be positive and is instead signed and negative. If you must assume signed or unsigned characters, comment them as SIGNED or UNSIGNED.

- Avoid assuming ASCII. If you must assume, document and localize. Remember that characters may hold (much) more than 8 bits.

- Code that takes advantage of the two's complement representation of numbers on most machines should not be used. Optimizations that replace arithmetic operations with equivalent shifting operations are particularly suspect. If absolutely necessary, machine-dependent code should be #ifdeffed or operations should be performed by #ifdeffed macros. You should weigh the time savings with the potential for obscure and difficult bugs when your code is moved.

- In general, if the word size or value range is important, typedef "sized" types. Large programs should have a central header file which supplies typedefs for commonly-used width-sensitive types, to make it easier to change them and to aid in finding width-sensitive code. Unsigned types other than unsigned int are highly compiler-dependent. If a simple loop counter is being used where either 16 or 32 bits will do, then use int, since it will get the most efficient (natural) unit for the current machine.

- Data *alignment* is also important. For instance, on various machines a 4-byte integer may start at any address, start only at an even address, or start only at a multiple-of-four address. Thus, a particular structure may have its elements at different offsets on different machines, even when given elements are the same size on all machines. Indeed, a structure of a 32-bit pointer and an 8-bit character may be 3 sizes on 3 different machines. As a corollary, pointers to objects may not be interchanged freely; saving an integer through a pointer to 4 bytes starting at an odd address will sometimes work, sometimes cause a core dump, and sometimes fail silently (clobbering other data in the process). Pointer-to-character is a particular trouble spot on machines which do not address to the byte. Alignment considerations and loader peculiarities make it very rash to assume that two consecutively-declared variables are together in memory, or that a variable of one type is aligned appropriately to be used as another type.

- The bytes of a word are of increasing significance with increasing address on machines such as the VAX (little-endian) and of decreasing significance with increasing address on other machines such as the 68000 (big-endian). Hence any code that depends on the left-right orientation of bits in a word deserves special scrutiny. Bit fields within structure members will only be portable so long as two separate fields are never concatenated and treated as a unit. [1,3] Actually, it is nonportable to concatenate *any* two variables.

- There may be unused holes in structures. Suspect unions used for type cheating. Specifically, a value should not be stored as one type and retrieved as another. An explicit tag field for unions may be useful.

- Different compilers use different conventions for returning structures. This causes a problem when libraries return structure values to code compiled with a different compiler. Structure pointers are not a problem.

- Do not make assumptions about the parameter passing mechanism, especially pointer sizes and parameter evaluation order, size, etc. The following code, for instance, is *very* nonportable.

```
        c = foo (*cp++, *cp++);
        char
foo (c1, c2, c3)
        char c1, c2, c3;
{
        char bar = *(&c1 + 1);
        return (bar);                   /* often won't return c2 */
}
```

This example has lots of problems. The stack may grow up or down (indeed, there need not even be a stack!). Parameters may be widened when they are passed, so a char might be passed as an int, for instance. Arguments may be pushed left-to-right, right-to-left, in arbitrary order, or passed in registers (not pushed at all). The order of evaluation may differ from the order in which they are pushed. One compiler may use several (incompatible) calling conventions.

- On some machines, the null character pointer ((char *)0) is treated the same way as a pointer to a null string. Do *not* depend on this.
- Do not modify string constants[7]. One particularly notorious (bad) example is

```
s = "/dev/tty??";
strcpy (&s[8], ttychars);
```

- The address space may have holes. Simply **computing** the address of an unallocated element in an array (before or after the actual storage of the array) may crash the program. If the address is used in a comparison, sometimes the program will run but clobber data, give wrong answers, or loop forever. The only exception is that a pointer into an array of objects may legally point to the first element after the end of the array. This "outside" pointer may not be dereferenced.
- Only the == and != comparisons are defined for all pointers of a given type. It is only portable to use <, <=, >, or >= to compare pointers when they both point in to (or to the first element after) the

[7]Some libraries attempt to modify and then restore read-only string variables. Programs sometimes won't port because of these broken libraries. The libraries are getting better.

same array. It is likewise only portable to use arithmetic operators on pointers that both point into the same array or the first element afterwards.

- Word size also affects shifts and masks. The following code will clear only the three right-most bits of an *int* on *some* 68000s. On other machines it will also clear the upper two bytes.

  ```
  x &= 0177770
  ```

  Use instead

  ```
  x &= ~07
  ```

  which works properly on all machines[8].

- Side effects within expressions can result in code whose semantics are compiler-dependent, since C's order of evaluation is explicitly undefined in most places. Notorious examples include the following.

  ```
  a[i] = b[i++];
  ```

  In the above example, we know only that the subscript into b has not been incremented. The index into a could be the value of i either before or after the increment.

  ```
  struct bar_t { struct bar_t *next; } bar;
  bar->next = bar = tmp;
  ```

  In the second example, the address of "bar->next" may be computed before the value is assigned to "bar". Compilers do differ.

- Be suspicious of numeric values appearing in the code ("magic numbers").

- Avoid preprocessor tricks. Tricks such as using /**/ for token pasting and macros that rely on argument string expansion will break reliably.

[8]The or operator (|) does not have these problems, nor do bitfields.

```
#define FOO(string) (printf("string = %s",(string)))
...
FOO(filename);
```

Will only sometimes be expanded to

```
(printf("filename = %s",(filename)))
```

Be aware, however, that tricky preprocessors may cause macros to break *accidentally* on some machines. Consider the following two versions of a macro.

```
#define LOOKUP(c)      (a['c'+(c)])        /* Sometimes breaks. */
#define LOOKUP(chr)    (a['c'+(chr)])      /* Works. */
```

The first version of LOOKUP can be expanded in two different ways and will cause code to break mysteriously.

- Become familiar with existing library functions and defines. (But not *too* familiar. The internal details of library facilities, as opposed to their external interfaces, are subject to change without warning. They are also often quite unportable.) You should not be writing your own string compare routine, terminal control routines, or making your own defines for system structures. “Rolling your own” wastes your time and makes your code less readable, because another reader has to figure out whether you’re doing something special in that reimplemented stuff to justify its existence. It also prevents your program from taking advantage of any microcode assists or other means of improving performance of system routines. Furthermore, it’s a fruitful source of bugs. If possible, be aware of the *differences* between the common libraries (such as ANSI, POSIX, and so on).
- Use *lint*[9]. It is a valuable tool for finding machine-dependent constructs as well as other inconsistencies or program bugs that pass the compiler. If your compiler has switches to turn on warnings, use them.
- Suspect labels inside blocks with the associated switch or goto outside the block.

[9] *Lint* is not available on many systems.

- Wherever the type is in doubt, parameters should be cast to the appropriate type. Always cast NULL when it appears in non-prototyped function calls. Do not use function calls as a place to do type cheating. C has confusing promotion rules, so be careful.
- Use explicit casts when doing arithmetic that mixes signed and unsigned values.
- The inter-procedural goto, longjmp, should be used with caution. Many implementations "forget" to restore values in registers. Declare critical values as volatile if you can or comment them as VOLATILE.
- Some linkers convert names to lower-case and some only recognize the first six letters as unique. Programs may break quietly on these systems.
- Beware of compiler extensions. If used, document and consider them as machine dependencies.
- A program cannot generally execute code in the data segment or write in to the code segment. Even when it can, there is no guarantee that it can do so reliably.

E.16 ANSI C

Modern C compilers support some or all of the ANSI proposed standard C. Write code to run under standard C whenever possible and use features such as function prototypes, constant storage, and volatile storage. Standard C improves program performance by giving better information to optimizers. Standard C improves portability by insuring that all compilers accept the same input language and by providing mechanisms that try to hide machine dependencies or emit warnings about code that may be machine-dependent.

E.16.1 Compatibility

Write code that is easy to port to older compilers. For instance, conditionally #define new (standard) keywords such as const and volatile in a global *.h* file. Standard compilers predefine the preprocessor symbol __STDC__. The void* type is hard to get right simply, since some older compilers understand void but not void*. It is easiest to create a new (machine- and compiler-dependent) VOIDP type, usually char* on older compilers.

```
#ifdef __STDC__
       typedef void *VOIDP;
#      define COMPILER_SELECTED
#endif
#ifdef A_TARGET
#      define const
#      define volatile
#      define void int
       typedef char *VOIDP;
#      define COMPILER_SELECTED
#endif
#ifdef ...
       ...
#endif
#ifdef COMPILER_SELECTED
#      undef COMPILER_SELECTED
#else
       { NO TARGET SELECTED! }
#endif
```

E.16.2 Formatting

The style for ANSI C is the same as for regular C, with two notable exceptions: storage qualifiers and parameter lists.

Because const and volatile have strange binding rules, each const or volatile object should have a separate declaration.

```
int const *s;          /* YES */
int const *s, *t;      /* NO */
```

Prototyped functions merge parameter declaration and definition in to one list. Parameters should be commented in the function comment.

```
/*
 *      ‘bp’: boat trying to get in.
 *      ‘stall’: a list of stalls, never NULL.
 *       returns stall number, 0 => no room.
 */
       int
enter_pier (boat_t const *bp, stall_t *stall)
```

```
{
        ...
```

E.16.3 Prototypes

Function prototypes should be used to make code more robust and to make it run faster. Unfortunately, the prototyped **declaration**

```
extern void bork (char c);
```

is incompatible with the **definition**

```
        void
bork (c)
        char c;
   ...
```

The prototype says that c is to be passed as the most natural type for the machine, probably a byte. The non-prototyped (backwards-compatible) definition implies that c is always passed as an int[10]. If a function has promotable parameters then the caller and callee must be compiled identically. Either both must use function prototypes or neither can use prototypes. The problem can be avoided if parameters are promoted when the program is designed. For example, bork can be defined to take an int parameter.

The above declaration works if the definition is prototyped.

```
        void
bork (char c)
{
        ...
```

Unfortunately, the prototyped syntax will cause non-ANSI compilers to reject the program.

It *is* easy to write external declarations that work with both prototyping and with older compilers[11].

[10]Such automatic type promotion is called widening. For older compilers, the widening rules require that all char and short parameters are passed as ints and that float parameters are passed as doubles.

[11]Note that using PROTO violates the rule "don't change the syntax via macro substitution." It is regrettable that there isn't a better solution.

```
#ifdef __STDC__
#      define PROTO(x) x
#else
#      define PROTO(x) ()
#endif

extern char **ncopies PROTO((char *s, short times));
```

Note that PROTO must be used with double parenthesis.

In the end, it may be best to write in only one style (e.g., with prototypes). When a non-prototyped version is needed, it is generated using an automatic conversion tool.

E.16.4 Pragmas

Pragmas are used to introduce machine-dependent code in a controlled way. Obviously, pragmas should be treated as machine dependencies. Unfortunately, the syntax of ANSI pragmas makes it impossible to isolate them in machine-dependent headers.

Pragmas are of two classes. Optimizations may safely be ignored. Pragmas that change the system behavior ("required pragmas") may not. Required pragmas should be #ifdeffed so that compilation will abort if no pragma is selected.

Two compilers may use a given pragma in two very different ways. For instance, one compiler may use "haggis" to signal an optimization. Another might use it to indicate that a given statement, if reached, should terminate the program. Thus, when pragmas are used, they must always be enclosed in machine-dependent #ifdefs. Pragmas must always be #ifdefed out for non-ANSI compilers. Be sure to indent the octothorpe (#) on the #pragma, as older preprocessors will halt on it otherwise.

```
#if defined(__STDC__) && defined(USE_HAGGIS_PRAGMA)
        #pragma (HAGGIS)
#endif
```

> "The '#pragma' command is specified in the ANSI standard to have an arbitrary implementation-defined effect. In the GNU C preprocessor, '#pragma' first attempts to run the game 'rogue'; if that fails, it tries to run the game 'hack'; if that fails, it tries to run GNU Emacs displaying the Tower of Hanoi; if that fails,

it reports a fatal error. In any case, preprocessing does not continue."

— *Manual for the GNU C preprocessor* for GNU CC 1.34.

E.17 Special Considerations

This section contains some miscellaneous do's and don'ts.

- Don't change syntax via macro substitution. It makes the program unintelligible to all but the perpetrator.

- Don't use floating-point variables where discrete values are needed. Using a float for a loop counter is a great way to shoot yourself in the foot. Always test floating-point numbers as <= or >=, never use an exact comparison (== or !=).

- Compilers have bugs. Common trouble spots include structure assignment and bitfields. You cannot generally predict which bugs a compiler has. You could write a program that avoids all constructs that are known broken on all compilers. You won't be able to write anything useful, you might still encounter bugs, and the compiler might get fixed in the meanwhile. Thus, you should write "around" compiler bugs only when you are forced to use a particular buggy compiler.

- Do not rely on automatic beautifiers. The main person who benefits from good program style is the programmer him/herself, and especially in the early design of handwritten algorithms or pseudo-code. Automatic beautifiers can only be applied to complete, syntactically correct programs and hence are not available when the need for attention to white space and indentation is greatest. Programmers can do a better job of making clear the complete visual layout of a function or file, with the normal attention to detail of a careful programmer (in other words, some of the visual layout is dictated by intent rather than syntax and beautifiers cannot read minds). Sloppy programmers should learn to be careful programmers instead of relying on a beautifier to make their code readable. Finally, since beautifiers are non-trivial programs that must parse the source, a sophisticated beautifier is not worth the benefits gained by such a program. Beautifiers are best for gross formatting of machine-generated code.

- Accidental omission of the second "=" of the logical compare is a problem. Use explicit tests. Avoid assignment with implicit test.

  ```
  abool = bbool;
  if (abool) { ...
  ```

  When embedded assignment is used, make the test explicit so that it doesn't get "fixed" later.

  ```
  while ((abool = bbool) != FALSE) { ...
  ```

  ```
  while (abool = bbool) { ... /* VALUSED */
  ```

  ```
  while (abool = bbool, abool) { ...
  ```

- Comment explicitly variables that are changed out of the normal control flow, or other code that is likely to break during maintenance.

- Modern compilers will put variables in registers automatically. Use the register sparingly to indicate the variables that you think are most critical. In extreme cases, mark the 2-4 most critical values as register and mark the rest as REGISTER. The latter can be #defined to register on those machines with many registers.

E.18 Lint

Lint is a C program checker [2] that examines C source files to detect and report type incompatibilities, inconsistencies between function definitions and calls, potential program bugs, etc. The use of *lint* on all programs is strongly recommended, and it is expected that most projects will require programs to use *lint* as part of the official acceptance procedure.

It should be noted that the best way to use *lint* is not as a barrier that must be overcome before official acceptance of a program, but rather as a tool to use during and after changes or additions to the code. *Lint* can find obscure bugs and insure portability before problems occur. Many messages from *lint* really do indicate something wrong. One fun story is about is about a program that was missing an argument to 'fprintf'.

```
 fprintf ("Usage: foo -bar <file>");
```

The *author* never had a problem. But the program dumped core every time an ordinary user made a mistake on the command line. Many versions of *lint* will catch this.

The -h, -p, -a, -x, and -c options are worth learning. All of them will complain about some legitimate things, but they will also pick up many botches. Note that -p checks function-call type-consistency for only a subset of Unix library routines, so programs should be linted both with and without -p for the best "coverage".

Lint also recognizes several special comments in the code. These comments both shut up *lint* when the code otherwise makes it complain, and they also document special code.

E.19 Make

One other very useful tool is *make* [7]. During development, *make* recompiles only those modules that have been changed since the last time *make* was used. Some common conventions include:

all	– always makes all binaries
clean	– remove all intermediate files
debug	– make a test binary 'a.out' or 'debug'
depend	– make transitive dep endencies
install	– install binaries
lint	– run lint
print/list	– make a hard copy of all source files
shar	– make a shar of all source files
spotless	– make clean, use revision control to put away sources.
	– Note: doesn't remove Makefile, although it is a source file
sources	– undo what spotless did
tags	– run ctags, (using the -t flag is suggested)
rdist	– distribute sources to other hosts
file.c	– check out the named file

In addition, command-line defines can be given to define either Makefile values (such as "CFLAGS") or values in the program (such as "DEBUG").

E.20 Project Dependent Standards

Individual projects may wish to establish additional standards beyond those given here. The following issues are some of those that should be addressed by each project program administration group.

- What additional naming conventions should be followed? In particular, systematic prefix conventions for functional grouping of global data and also for structure or union member names can be useful.
- What kind of include file organization is appropriate for the project's particular data hierarchy?
- What procedures should be established for reviewing *lint* complaints? A tolerance level needs to be established in concert with the *lint* options to prevent unimportant complaints from hiding complaints about real bugs or inconsistencies.
- If a project establishes its own archive libraries, it should plan on supplying a lint library file [2] to the system administrators. The lint library file allows *lint* to check for compatible use of library functions.
- What kind of revision control needs to be used?

E.21 Conclusion

A set of standards has been presented for C programming style. Among the most important points are:

- The proper use of white space and comments so that the structure of the program is evident from the layout of the code. The use of simple expressions, statements, and functions so that they may be understood easily.
- To keep in mind that you or someone else will likely be asked to modify code or make it run on a different machine sometime in the future. Craft code so that it is portable to obscure machines. Localize optimizations since they are often confusing and may be "pessimizations" on other machines.
- Many style choices are arbitrary. Having a style that is consistent (particularly with group standards) is more important than following

absolute style rules. Mixing styles is worse than using any single bad style.

As with any standard, it must be followed if it is to be useful. If you have trouble following any of these standards don't just ignore them. Talk with your local guru, or an experienced programmer at your institution.

E.22 References

1. B.A. Tague, *C Language Portability*, Sept 22, 1977. This document issued by department 8234 contains three memos by R.C. Haight, A.L. Glasser, and T.L. Lyon dealing with style and portability.

2. S.C. Johnson, *Lint, a C Program Checker*, USENIX UNIX Supplementary Documents, November 1986.

3. R.W. Mitze, *The 3B/PDP-11 Swabbing Problem*, Memorandum for File, 1273-770907 .01MF, September 14, 1977.

4. R.A. Elliott and D.C. Pfeffer, *3B Processor Common Diagnostic Standards – Version 1*, Memorandum for File, 5514-780330.01MF, March 30, 1978.

5. R.W. Mitze, *An Overview of C Compilation of UNIX User Processes on the 3B*, Memorandum for File, 5521-780329.02MF, March 29, 1978.

6. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

7. S.I. Feldman, *Make: A Program for Maintaining Computer Programs*, USENIX UNIX Supplementary Documents, November 1986.

8. Ian Darwin and Geoff Collyer, Can't Happen or /* NOTREACHED */ or Real Programs Dump Core, USENIX Association Winter Conference, Dallas 1985 Proceedings.

9. Brian W. Kernighan and P. J. Plaugher *The Elements of Programming Style*, McGraw-Hill, 1974.

10. J. E. Lapin, *Portable C and Unix System Programming*, Prentice-Hall, 1987.

E.23 Stuff to Remember

The Ten Commandments for C Programmers

Henry Spencer

1. Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgement oft exceed thine.
2. Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.
3. Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.
4. If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.
5. Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".
6. If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.
7. Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.
8. Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.
9. Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity

stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.

10. Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX", and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.