



Notes on C

Table of Contents

Preface	vii
1. Latest (last?) version	vii
1. Tutorial Introduction to C	1
1.1. Software and Computer Languages	1
1.2. Evolution of C	1
1.3. Warning: C may be “dangerous”	2
1.4. Overview of Basic C	2
1.5. Examples	3
1.6. Some details	7
1.6.1. Using printf() formatting	7
1.6.2. Character Escape Sequences	8
1.7. Using functions	8
1.7.1. Centimetres to Inches (version 2)	9
1.8. Simple data conversion (Filters)	9
1.8.1. Copying input to output	10
1.8.2. Translating lower case to upper case	11
1.9. Casts	12
1.10. I/O redirection and piping	12
1.10.1. Piping	13
2. Basic C Syntax	15
2.1. Fundamental Data Types	15
2.2. Declarations	17
2.2.1. Enumerated (enum) types	17
2.3. Constants	18
2.3.1. Examples	18
2.4. Operators and Expressions	19
2.4.1. Arithmetic operators	19
2.4.1.1. Differences between pre- and post- operators	19
2.4.2. Logical Operators	20
2.4.3. The assignment operator	21
2.4.4. The sizeof operator	21
2.4.4.1. Example	21
2.5. Simple & block statements	22
2.6. Flow Control	22
2.6.1. if statement	22
2.6.2. if...else statement	23
2.6.3. While loop	23
2.6.4. Expressions in conditionals	23
2.6.5. Do ... while loop	24
2.6.6. For loop	25
2.6.6.1. Examples	25
2.6.6.2. Break and Continue statements	26

2.6.6.3. Switch statement	27
2.7. Bit Operators	28
2.7.1. Examples	29
2.8. Expanded Assignment Statements	29
2.9. Conditional (?) operator	30
2.10. The Comma (,) operator	30
3. Project management and make	33
3.1. Using separate source code files	33
3.2. Scope of variable names	33
3.2.1. Notes	34
3.3. Make	35
3.3.1. A Simple Example	35
3.3.2. Compilation example	36
3.3.3. Additional remarks	37
3.3.4. Using generic rules	38
3.3.5. Parting remarks	39
4. Basic Arrays and Pointers	41
4.1. Arrays	41
4.2. Pointers	41
4.2.1. The & Operator	42
4.2.2. Pointer Arithmetic	42
4.2.2.1. Strings are character pointers	43
4.3. Arrays and Pointers	44
4.4. Functions	45
4.4.1. Examples	46
4.4.2. Differences between ANSI and K&R C	46
4.5. Examples of pointers	47
4.5.1. Comamnd line arguments	47
4.5.2. Variable Number of Arguments	49
4.5.2.1. stdarg() right way to handle variable number of arguments	51
5. Pointers to functions	53
5.1. Complex declarations	55
6. Parsing	57
6.1. Compilers: An Overview	57
6.1.1. Lexical analysis	58
6.1.2. Parsing	59
6.1.2.1. Parsing C declarations	66
7. Data Structures	75
7.1. Structures and Unions	75
7.2. Structure Examples	75
7.3. Using typedef	76
7.4. Pointers to structures	77
7.5. Single Linked List Example	78
7.6. Initializing Linked lists in declarations	78
7.7. Unions	79

7.8. Example: Doubly Linked List	80
7.8.1. Header file doubleLinkedList.h	81
7.8.2. Main routine	81
7.8.3. Print list	83
7.8.4. Notes	84
8. The Preprocessor	85
8.1. Using the Preprocessor	85
8.1.1. #include	85
8.1.2. #define	86
8.1.3. #ifdef and #ifndef	87
8.1.4. #undef	88
8.1.5. ANSI C preprocessor	89
9. The Standard library	91
9.1. Strings	91
9.2. ctype.h	91
9.2.1. Implementation of ctype.h	92
9.3. stdio.h	93
9.3.1. File I/O	94
9.3.1.1. Opening a file	94
9.3.1.2. Writing to a file	95
9.3.1.3. Reading a file	95
9.3.1.4. Closing a file	95
9.4. assert.h	95
9.5. Memory	96
9.6. stdarg.h	97
9.7. Handling errors	97
10. Data-driven Programming	99
10.1. Example—Finite State Machine	99
10.1.1. The BAD Way	100
10.1.2. A BETTER Way	102
10.1.3. Doing It All Automatically	104
10.1.3.1. Example	106
10.1.3.2. Notes	107
10.2. Example—FSM and function pointers	108
10.3. Menu Driven Code	109
Bibliography	111

Preface

The original version of these “Notes on C” were written in the mid 1980’s as a set of slides for a seminar I gave on C programming.

Over the years, I referred students to the “Notes on C” for various courses. I also updated the original notes 3 or 4 times. With each update, the original slides became more like notes (and, sometimes, even had a book-like quality).

However, since I only updated sections in a haphazard way, the result was not as uniform and consistent as I would have liked.

1. Latest (last?) version

In January, 2005, I started to revise the whole thing in two ways:

- Change the markup language I used to author the notes from LaTeX to XML (using the Docbook DTD).
- Make minor content changes to make the notes more uniform and to eliminate some references that are no longer necessary.

The results?

Well, the conversion from LaTeX to XML was far more time-consuming and tedious than I had anticipated. Nonetheless, the (almost) total separation of logical content and form (rendition) has considerable advantages.

The content did not change very much. I eliminated many (but not all) comparisons between K&R C and ANSI/ISO C. (The original slides were written when ANSI C was a mature proposal but not yet official.) I also removed many comparisons between C and Pascal (which had been a popular language in the 1980s)

There remain, of course, many warts, flaws and out-and-out errors in these Notes. Nonetheless, I want to move on: let the warts and flaws remain. I will, however, fix out-and-out errors that are brought to my attention. (Note: by out-and-out error, I mean something that is in clear contradiction with the ANSI/ISO C Standard. I do not mean content that is “unclear”, “ambiguous”, “ugly”, “incomplete”, etc.)

Chapter 1. Tutorial Introduction to C

1.1. Software and Computer Languages

Table 1.1. Programming Languages

LANGUAGE	APPLICATION
Assembler	Low level programming of small applications on 8-bit controllers.
C	Simple systems programming language allowing access to underlying machine features.
C++	Object oriented extension to C.
Java	An object-oriented language with C-like syntax. A very portable language.

1.2. Evolution of C

1. B language (based on BCPL) written by Dennis Ritchie at Bell Labs in early '70s as first pass in writing UNIX in a high level language.
2. B was a typeless language (it accessed machine data like bytes and words without reference to the interpretation of the contents; i.e. 32-bit integers or 32-bit floating point numbers were simply referred to as *words*).
3. B was soon transformed into a weakly typed language: C.
4. C was defined in the first edition (1978) of the Kernighan and Ritchie book *The C Programming Language* [KandR78](#). We will refer to this as *traditional C*. (It is also called *K&R C*.)
5. The syntax of traditional C was defined more precisely by ANSI (American National Standards Institute).
6. Unless otherwise stated, we use ANSI C here.
7. ANSI defines two standards in C:

Freestanding C

Describes the *pure* language itself.

Standard Library

Defines a set of functions and definitions that must be supplied by a compiler in order to be fully ANSI-compliant.

8. The C++ language incorporates object oriented programming practices.

1.3. Warning: C may be “dangerous”

1. The designers of C wanted to use the language for systems programming. Consequently, they required:
 - C be sufficiently powerful to access many aspects of the underlying hardware and do things that would normally require assembly language.
 - C produce very efficient code.
2. For example, C can treat data objects as just “bit patterns” in the same way that an assembly language programmer can. Similarly, a C programmer can manipulate an address like an “ordinary number” and then access the memory location(s) referenced by this “number” in any way he or she sees fit. While these features are desirable, they come at a cost. Most importantly:
 - There is *no run time checking* in C. Specifically:
 - Overflow of integer values is not detected at run time.
 - There is no run-time checking of array indices to see if they are within the declared bounds of the array. Access beyond the limits of an array will not cause a run-time error in an unprotected OS such as MS-DOS. It may or may not result in a segmentation violation in a protected OS such as Windows/NT or UNIX.
 - Memory references can point anywhere—even where they should not point!
 - C programmer's can use *type casting* to modify the data type of an object at run time. There is scant checking that such type casting makes any sense.

1.4. Overview of Basic C

A C program consists of:

- Zero or more declarations of global variables (defined outside of any functions).
- One or more functions consisting of a *name*, a (possibly empty) *list of arguments* enclosed in parenthesis, and a *function body* consisting of a *compound statement*.
- A *compound statement* is enclosed in curly braces and consists of:
 - Zero or more declarations of *local variables* that exist only within the compound statement.
 - Zero or more *statements*. A statement is either:

1.5. Examples

- Another compound statement; or,
- A simple statement (e.g. assignment) terminated with a semi-colon.

- *Preprocessor directives* (which begin with a # and are transformed into “raw C” by the preprocessor before the C compiler actually looks at the source code.
- Finally, a C program can have any number of *comments* delimited by /* and */.

-  • There are only *functions* in C. There are no subroutine or program blocks. (However, the function called `main(...)` will, under most OS's, define the entry point to the program. The `main()` function returns an integer value that the OS usually interprets as a return code.

- C is **not** a block-structured language like Pascal or Ada. (i.e. Functions *cannot* be declared inside a function.)

1.5. Examples

We now informally elaborate on these points with some sample programs.

Example 1.1. hello.c

The first program simply outputs the message Hello world!.

```
/*
   First C Program: hello.c
   Displays message: Hello, world!
*/

int main()
{
    printf("Hello, world!\n");
    return 0; /* or, better, exit(0); */
}
```

(The source code file [hello.c](#) [./src/hello.c] is available.)

The main function block consists of 2 simple statements: The first statement uses the *standard library* `printf(...)` function. Its argument is a string of characters which is output.

The return statement terminates the main function returning the value of 0 (ZERO) to the caller, in this case the operating system. In any function, the `return returnValue` statement will return to the caller; however, the `exit(int returnValue)` statement will always terminate the entire program and return its `returnValue` to the OS.

For the `main()` function, there is no difference between `return(0)` and `exit(0)`.

Example 1.2. A Simple loop

The following simple program prints out a table converting inches to centimetres.

(The source code file `lengthConversion.c` [`./src/lengthConversion.c`] is available.)

```
/**
 * Summary: Prints a table of Centimetres vs. Inches
 */
#include <stdlib.h> ❶
#define CM_PER_INCH 2.54 ❷
#define SMALL_LENGTH_CM 1.0
#define BIG_LENGTH_CM 4.0
#define DIFF_LENGTH 0.5
int main()
{
    double cm, inch;❸

    cm = SMALL_LENGTH_CM;

    printf("Centimetres   Inches\n");
    printf("_____ \n");
    while(cm <= BIG_LENGTH_CM) {❹
        inch = cm / CM_PER_INCH;
        printf("  %4.1f      %4.2f\n", cm, inch);❺
        cm = cm + DIFF_LENGTH;
    }
    exit(0);
}
```

- ❶ The `#include <stdlib.h>` statement is a *pre-processor directive* to insert the named file into the source code. The particular file included (`stdlib.h`) contains definitions of constants and other statements required when certain standard functions are used. In this case, it is required so that the `exit()` function is understood.
- ❷ The `#define` statements are interpreted by the preprocessor so that subsequent use of the defined name is replaced by the defined value.

Thus, following `#define BIG_LENGTH_CM 5.0`, every occurrence of `BIG_LENGTH_CM` is replaced by the constant `5.0`.

This practice of using symbolic constants is strongly recommended for several reasons:

- The program is more readable.

- “Magic numbers” are isolated in the `#define` statements. Often, the overall behavior of a program is very dependent on the specific values of these magic numbers. When they are isolated in `#define` statements at the beginning of a program, the program's behavior can be modified simply by changing the defined values. With good symbolic names and comments, it is often possible to make this kind of change to a program without understanding or even looking at the actual code.
- When the same magic number appears more than once in a program, it is less likely that the wrong value will be typed in due to a typographical error. Furthermore, it becomes much easier and less error-prone to change the magic number by editing the `#define` instead of slogging through the source code to find the instances of the number.

- ③ All variables must be *declared* in C before they are used. The declaration “`double cm, inch`” declares the variables `inch` and `cm` to be *doubles* (i.e. floating point numbers).
- ④ The body of the `while` loop is executed unless the condition tested for in the `while(condition)` statement is false. The test is made before the body is executed and again before executing the body another time. (Hence, if the condition is false at the outset, the loop body is not executed at all.)

In this case, the loop body is executed if `cm` is less than or equal to the magic number `BIG_LENGTH_CM`.

The `cm` variable is incremented at the end of the loop and the loop executed again if the `while` condition is still true.

In this case, the loop body is executed 7 times for the values of 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, and 4.0.

- ⑤ The statement

```
printf("  %4.1f      %4.2f\n", cm, inch);
```

in the `while` loop contains formatting commands introduced by a `%` in the first argument.

Previous `printf()` statements have simply echoed the formatting string to the output. The `%` character indicates that another argument has been passed to `printf`; the character(s) following the ``%'` indicate how the argument is to be printed out.

In this case, the “`%4.1f`” formatting command indicates that the argument is a floating point number, that it is to be printed in at least 4 columns and that one digit after the decimal point should be printed.

1.6.1. Using printf() formatting

The output of this program is:

```
Centimetres  Inches
-----
  1.0         0.39
  1.5         0.59
  2.0         0.79
  2.5         0.98
  3.0         1.18
  3.5         1.38
  4.0         1.57
```

1.6. Some details

1.6.1. Using printf() formatting

The following table shows the most common formatting commands used in `printf()` formatting strings.

Table 1.2. Printf formatting

Format	Meaning
<code>%d</code>	integer as a signed decimal number
<code>%x</code>	integer as an unsigned hexadecimal number
<code>%o</code>	integer as an unsigned octal number
<code>%u</code>	integer as an unsigned decimal number
<code>%c</code>	a character
<code>%s</code>	a string
<code>%f</code>	a floating point number
<code>%e</code>	a floating point number
<code>%%</code>	a literal <code>'%'</code> character
<code>%e</code>	a floating point number
<code>%g</code>	a floating point number

Between the `%` and the data type specifier, a number can be used to specify the minimum space allocated for the result. Thus `"%4d"` would print the integer 23 with 2 leading spaces. (The width qualifier is a minimum width: thus `"%2d"` used to print the integer 123 would not chop off the leading 1; the width would be increased to 3 to accommodate all the digits.)

1.6.2. Character Escape Sequences

The table below gives common escape sequences to specify special characters in strings.

Table 1.3. Special characters in strings

Notation	Character
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash
\"	Double quote
\'	Single quote
\ddd	Octal representation
\xxx	Hex representation

1.7. Using functions

Functions can be used and defined in C to return a value dependent on the arguments given.

For example, we could define a function to convert centimetres to inches as follows:

(The source code file [cm2inch.c](#) [./src/cm2inch.c] is available.)

```

/*
 * double cm2inch(double cm)
 *
 * SUMMARY:
 *   Description: Converts cm to inches
 */

#define CM_PER_INCH 2.54

double cm2inch(double cm) ❶
{
    return cm * CM_PER_INCH;
}

```

1.8. Simple data conversion (Filters)

- ❶ The line `double cm2inch(double cm)` declares the function as returning a double-precision floating point value and taking a double parameter, called `cm`, as an argument.

☞ This example is a bit “over the top”. Typically, such a simple calculation would be directly embedded in the source code (or defined as a *macro*).

1.7.1. Centimetres to Inches (version 2)

We can now re-write the `main()` of the original program as shown below.

(The source code file [lengthConversionV2.c](#) [./src/lengthConversionV2.c] is available.)

```
/**
 * Summary: Prints a table of Centimetres vs. Inches
 */
#include <stdlib.h>
#include <stdio.h>
double cm2inch(double ); ❶
#define BIG_LENGTH_CM 4.0
#define DIFF_LENGTH 0.5
int main()
{
    double cm;

    cm = SMALL_LENGTH_CM;

    printf("Centimetres   Inches\n");
    printf("_____ \n");
    while(cm <= BIG_LENGTH_CM) {
        printf("  %4.1f      %4.2f\n", cm, cm2inch(cm));
        cm = cm + DIFF_LENGTH;
    }
    exit(0);
}
```

- ❶ The `double cm2inch(double);` statement is a *function prototype* informing the compiler the data type returned and the type(s) of the passed parameter(s).

1.8. Simple data conversion (Filters)

One of the most common idioms in C programs is to read input one character at a time, perform some processing on it, and output it one character at a time.

1.8.1. Copying input to output

In the simplest filter program, we simply copy the input to the output. (The source code file `copyin.c` [./src/copyin.c] is available.)

```
/*
 * Copyin copies its input to the output one character
 * at a time.
 */

#include <stdio.h>
main()
{
    int ch; ❶

    ch = getchar(); /* get next input character */
    /* loop until "End Of File" is reached */
    while (ch != EOF) {
        putchar(ch); /* Output the character */
        ch = getchar(); /* Read next character */
    }
    exit(0);
}
```

- ❶ • Note that `ch` (the return value of `getchar()`) is declared as an `int`, **not** a `char`.
- Why?
- The `getchar()` returns an integer not a character. In this way, a special integer value that does not correspond to any actual character can be used to signal end-of-file.

The special value is defined in `stdio.h` as the symbolic constant `EOF`. (Note that `EOF` is almost always defined as being `-1`. However, you should not rely on this.)

The `while` condition “`(ch != EOF)`” means that the body should be executed if the end-of-file has not been reached.

The function `putchar(ch)` outputs the character given as an argument.

It is good practice to make the last statement in a program the `exit(value)` statement: a value of zero should be used for a program that terminated normally; any other value indicates abnormal termination.

1.8.2. Translating lower case to upper case

A slightly more complex example involves converting the input characters from lower to upper case.

(The source code file `toupper.c` [./src/toupper.c] is available.)

```
#include <stdio.h>
main()
{
    int ch;
    ch = getchar();
    while ( ch != EOF) {
        if ( ch >= 'a' && ch <= 'z' ) /* if lower case */ ❶
            ch = ch + 'A' - 'a'; /* convert it to upper */ ❷
        putchar(ch);
        ch = getchar();
    }
}
```

- ❶ This if statement is true if `ch` lies in the range of characters between 'a' and 'z'.
- ❷ Conceptually, the character is converted to upper case by subtracting the code for an 'a' (yielding an integer between 0 and 25) and then adding the code for an 'A'. Note that the expression `'A' - 'a'` is a constant (with the value 32 for ASCII character encoding) which will be evaluated at compile time, not at run time. Note also that writing the code this way makes it independent of the particular character encoding used (eg. ASCII vs. EBCDIC).

 Note that the above program could be written more cleanly as:

```
#include <stdio.h>
main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        if ( ch >= 'a' && ch <= 'z' )
            ch = ch + 'A' - 'a';
        putchar(ch);
    }
}
```

Finally, an experienced C programmer would probably write: (The source code file `toupper3.c` [./src/toupper3.c] is available.)

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

```

main()
{
    int ch;
    while ((ch = getchar()) != EOF) {
        putchar(islower(ch) ? ch + 'A' - 'a' : ch);
        /* OR simply: putchar(toupper(ch)); */
    }
}

```

When we look at C in more detail later on, we will see how these more efficient versions work.

1.9. Casts

- Once a variable has been declared, it is possible to force it to another type with a *cast*.
- The coercion to another type is done by preceding the expression with the name of data type we want in parenthesis.
- For example, to print out the integer *i* as a floating point number, use:

```
printf("%f\n", (float) i);
```

- Without the cast, the output would be garbage.
- Casts are especially useful when *pointers* are used as we shall see when we look at pointer variables.

1.10. I/O redirection and piping

When we talk about “input” and “output”, we should really say *stdin* (for *standard input*) and *stdout* (for *standard output*). The C library functions like `printf()`, `putchar()` and `getchar()` actually manipulate the “files” *stdin* or *stdout* and it is the Operating System that defines what these “files” are.

Under both UNIX and Microsoft operating systems, *stdin* is by default the keyboard input and *stdout* is the screen. However, both Operating Systems allow *stdin* and/or *stdout* to be redefined at the time the command is invoked. On the command line, `< file_name_in` is used to redefine *stdin* and `> file_name_out` redefines *stdout*.

Thus, assuming the command **copyin** copies *stdin* to *stdout*, we can write:

```
copyin < copyin.c > copyin.bak
```

to copy the file `copyin.c` to `copyin.bak`.

1.10.1. Piping

Similarly:

```
upl < copyin.c > copyin.up
```

creates a copy of `copyin.c` with all the lower case letters converted to upper case in the file `copyin.up`

With this useful trick, we can avoid learning about how to use files until later on.

1.10.1. Piping

Both UNIX and Windows also allow two commands to be joined together so that *stdout* of one becomes *stdin* of the second. This is called a *pipeline* and the ``|'` character (vertical bar) is used to specify a pipeline.

For example:

```
lengthConversion | toupper3
```

directs the output of the **lengthConversion** command into the input of the **toupper3** command. The result is that all the lower case letters in the normal output of **lengthConversion** are converted to upper case.

We can combine these ideas with:

```
lengthConversion | toupper3 > lengthConversion.out
```


Chapter 2. Basic C Syntax

2.1. Fundamental Data Types

Integers (`int`):

- Up to 3 sizes of integers: `short`, `long` and default `int`;
- On most systems: `long` and `int` are 32 bits; `short` is 16 bits.
- Integers can be signed or unsigned. For n-bit (signed) integers,

$$-2^{n-1} \leq \text{int} \leq 2^{n-1}-1$$

For n-bit unsigned integers,

$$0 \leq \text{unsigned} \leq 2^n-1$$

Example 2.1. All one's bit pattern

The bit pattern 1111...11 represents -1 if interpreted as a signed number or 2^n-1 if interpreted as unsigned. (We assume that signed numbers use the *two's complement* convention which is "almost" universally adopted.)

Thus the statement:

```
printf("-1 (signed): %d; -1 (unsigned) %u, -1 (hex) 0x%x\n", -1, -1, -1);
```

produces the output (on a machine where ints are 32 bits):

```
-1 (signed): -1; -1 (unsigned) 4294967295, -1 (hex) 0xffffffff
```

(The source code file [printfMinusOne.c](#) [./src/printfMinusOne.c] is available.)

Character (`char`):

8 bits. Usually interpreted as signed. (ANSI C allows specification of signed or unsigned characters.)

Reals

Real numbers are (potentially) inexact. C provides two generic types: `float` and `double`.

Most compilers use the IEEE standard to represent real numbers. Floats are usually 32 bits, doubles 64 bits.

 There is no boolean type. Integers are used instead. False is represented as 0; true as any other value.

The table below summarizes the primitive data types.

Table 2.1. Fundamental Type Summary¹

Name	Size (bits)	Range
int	32	$-2^{31} \dots 2^{31} - 1$
long	32	$-2^{31} \dots 2^{31} - 1$
short	16	$-2^{15} \dots 2^{15} - 1$
unsigned	32	$0 \dots 2^{32} - 1$
unsigned long	32	$0 \dots 2^{32} - 1$
unsigned short	16	$0 \dots 2^{16} - 1$
unsigned char	8	$0 \dots 2^8 - 1$
char	8	$-2^7 \dots 2^7 - 1$
float	32	$-1.2 \times 10^{-38} \dots 3.4 \times 10^{38}$
double	64	$-2.2 \times 10^{-308} \dots 1.8 \times 10^{308}$

¹ This table represents typical sizes for 32-bit machines. Note, however, that smaller machines often use 16-bit integers; 64-bit machines usually have 64-bit longs.

2.2. Declarations

Example 2.2. Declarations

```
int i, j, k; ❶  
short ii, jj; ❶  
unsigned short lll_5; ❶  
char ch, input_char; ❶  
double big_or_small; ❶  
int aVeryLong_NameMoreThanThirtyTwoCharacters; ❶ ❷  
unsigned _foo; ❶ ❸  
char CH; ❶ ❹
```

- ❶ Note that characters in names can be any letter, an underscore (`_`), or a digit but cannot begin with a digit.
- ❷ Names are sometimes limited to 16 or 32 characters.
- ❸ Names *may* begin with an underscore; however, by convention only standard library or Operating System functions do this. **Do not** begin a name with an underscore unless you are sure you know what you are doing.
- ❹ Similarly, you *may* use only upper case letters in a name; again, by *convention*, upper-case only names are usually used only for symbolic constants.

2.2.1. Enumerated (enum) types

- The enumerated type specifies a subset of integers with each member having a symbolic name.
- For example, we can write:

```
enum Colours {RED, GREEN, BLUE};
```

to define the *enumerated type* Colours which has three symbolic values: RED, GREEN and BLUE.

- We can now declare arguments of type Colours:

```
Colours colour1, colour2, colour3;
```

- The variables colour1, colour2, and colour3 can take on the symbolic values associated with the Colour (enum) type. Also, since these are ultimately integers, normal integer arithmetic can be performed on them. Thus statements like:

```
colour2 = RED;
colour3 = colour2++;
```

are legal.

- Note, however, that C does no run time checking to verify that the result of the arithmetic is a valid integer for the enum type. (i.e. it is not as safe as the *pascal* equivalent.)
- By default, the first symbolic constant will be given the integer value of zero; each additional constant is converted to the next integer.
- The enumerated type is a useful alternative to #define preprocessor directives. For example:

```
enum ERROR_CODES {
    ERROR_NO_MEMORY,
    ERROR_BAD_INPUT_FILE,
    ERROR_BAD_OUTPUT_FILE,
    NUM_ERROR_CODES
};
```

- Note that in the above enum, the integer value of NUM_ERROR_CODES would be 3—i.e. the actual number of error codes enumerated.

2.3. Constants

- Integer constants can be written in the default decimal notation (eg. 123), as octal numbers by using a leading 0 (eg. 0123 (octal) is the same number as 83 (decimal), or as hex numbers by using a leading 0x (e.g. 0x123 = 291 (decimal) = 0443 (octal)).
- Long integer constants are specified by appending an l or L to the integer. (e.g. 123l or 123L). (Note that L is preferred: using l is easily confused with the digit 1).
- Character constants are indicated by putting the character in single quotes (e.g. 'a' for the character code for the letter *a*).

2.3.1. Examples

```
long longNumber; char singleCharacter;
longNumber = 123L; singleCharacter = 'a';
```

2.4. Operators and Expressions

2.4.1. Arithmetic operators

The following arithmetic operators are available:

Table 2.2. Arithmetic operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (ints only)
<code>++var</code>	preincrement
<code>var++</code>	postincrement
<code>--var</code>	predecrement
<code>var--</code>	postdecrement

 Note that there is no exponentiation operator.

2.4.1.1. Differences between pre- and post- operators

- There is no difference between the statements `++x;` and `x++;`. They both simply increment the value of `x` by 1.
- However, consider the following sequences:

```
x = 1;
y = x++; /* or y = ++x; */
printf("y: %d, x: %d\n", y, x);
```

When we use `y = x++;`, `y` is assigned the value `x` had *before* being incremented: `x` is incremented after its value is used in the expression. Thus the values printed for `y` and `x` would be 1 and 2.

If the commented-out statement `y = ++x;` were used instead, `x` is incremented before being used in the expression. Hence both `x` and `y` would have the value of 2 when printed.

2.4.2. Logical Operators

The following table lists the logical operators that can be used.

Table 2.3. Logical operators

Operator	Description
==	equals
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
	logical <i>or</i>
&&	logical <i>and</i>
!	logical <i>not</i>

Note that since there is no Boolean type, the result of a logical operation is an integer: 1 if the condition is true; 0 if false. (While it is true that, in general, *any* non-zero integer represents **true**, the specific value of 1 is used to represent a true result of these logical operators.)

Hence, the following is legal:

```
int n;

n = (i > 5) + (j > 10);
```

In the above case, n will be zero only if both conditions fail; if exactly one is true, n will be 1 (but we don't know which one is true); if they are both true, n will be 2.

Note that when logical expressions are joined by the logical *and* (&&), they are evaluated left to right until an expression becomes false. Hence the following (which would lead to disaster in a language like *pascal*) is legal and safe:

```
if (i != 0 && j/i > 10)
```

(i.e. If i is zero, the first test will fail and the second test, involving division by i , will not be done, hence avoiding a “divide-by-zero” error.)

2.4.4. The sizeof operator

Similarly, expressions joined with the logical *or* (`|`) are evaluated left to right until one evaluates as true.

2.4.3. The assignment operator

- The *assignment operator* sets its left side to the value of the right side expression. This value is also the value of the *assignment expression*

Hence, the following are valid statements:

```
i = j = 5;
i = (j = 3) + (k = 2);
```

- The `i = (j = 3) + (k = 2);` statement is equivalent to:

```
j = 3;
k = 2;
i = j + k;
```

2.4.4. The sizeof operator

The **sizeof** (type name or variable) gives the number of bytes for the type.

2.4.4.1. Example

```
int i;
char c;
long l;
.
.
printf("On this machine, ints are %d bytes"
      "chars are %d bytes, and"
      "longs are %d bytes\n",
      sizeof(int),
      sizeof char,
      sizeof(l));
```



- We exploit the C preprocessor's ability to concatenate strings separated by whitespace into a single string in the above example.
- The alternative (in K&C) would be:

```
printf("On this machine, ints are %d bytes\
      chars are %d bytes, and\
      longs are %d bytes\n");
```

- Here, we use the backslash (\) to write a long string across several lines for increased readability.
- Despite appearances, `sizeof` is an *operator*, not a function. That is why `sizeof char` is legal. The parenthesis around the operand are normally added for readability.

2.5. Simple & block statements

1. A *statement* can be either a *simple* statement or a *compound* statement.
2. The most elementary form of a simple statement is any expression followed by a semi-colon. Thus:

```
x = i;          /* A simple statement */
```

3. The other forms of simple statements include things like the *if statement* to be discussed below.
4. A compound statement is any number of of statements bracketed by braces ({...}). Thus:

```
{ /* Compound statement enclosed with braces {} */  
  x = j++;  
  i++;  
}
```

2.6. Flow Control

We now examine the precise syntax of the flow control statements including:

- **if** statement;
- **if ... else** statement;
- **while** statement;
- **do** statement;
- **for** statement;
- **switch** statement;

2.6.1. if statement

Given:

```
if (expression)
```

2.6.4. Expressions in conditionals

```
statement
```

The *statement* (simple or compound) is executed only if *expression* is true (i.e. evaluates to non-zero).

2.6.2. if...else statement

Given:

```
if (expression)
    then_statement
else
    else_statement
```

The *then_statement* (simple or compound) is executed if *expression* is true (i.e. evaluates to non-zero); otherwise, the *else_statement* is executed.

2.6.3. While loop

The form:

```
while (expression)
    statement
```

is equivalent to:

```
label: if (expression)
    {
        statement
        goto label;
    }
```

i.e. the body of the loop will be executed 0 or more times until the expression becomes false.

2.6.4. Expressions in conditionals

1. Note that the conditional expression can be *anything*. If the value of the expression is zero, the condition is false; otherwise it is true.
2. Hence the following:

```
while (c = getchar())
```

```
putchar(c);
```

will copy *stdin* to *stdout* until a null character (ascii 0) is read.

3. An even more common idiom is:

```
while ((c = getchar()) != EOF)
    putchar(c);
```

The above program is just like the *copyin* program we saw earlier. Note that the parenthesis are necessary to force the assignment to occur before the inequality test. If we wrote:

```
if (c = getchar() != EOF)
```

it would be interpreted as:

```
if (c = (getchar() != EOF))
```

This would result in *c* being assigned the value *true* (1) or *false* (0), not the value of *c* read in!

2.6.5. Do ... while loop

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

is equivalent to:

```
label:    statement
        if (expression) goto label
```

i.e. the body of the loop will be executed 1 or more times until the expression becomes false.

Note that *while* loops are safer than *do* loops since the former execute 0 or more times but the latter executes at least once. (The *do* loops may be more efficient, however.) Neglecting to account for the possibility that a loop should not be executed at all can lead to catastrophic program failure and we strongly encourage the use of *while* instead of *do*.

To understand the differences, consider the following:

2.6.6. For loop

```
    /* DO version */
do
{
    /* launch nuclear missiles */
} while ( /* enemy is attacking us */ )

    /* WHILE version */
while ( /* enemy is attacking us */ )
{
    /* launch nuclear missiles */
}
```

The *do* version, of course, would cause the immediate, inevitable beginning of a nuclear war!

2.6.6. For loop

```
for (expr1; expr2; expr3)
    statement
```

is equivalent to:

```
expr1;          /* i.e. expr1 is done before entering loop */
while (expr2)
{
    /* loop if expr2 is non-zero */
    statement    /* loop body proper */
    expr3;       /* re-initialization */
}
```

 All components (*expr1*, *expr2*, and *expr3*) are optional.

2.6.6.1. Examples

```
    /* Infinite loop */
for ( ; ; )
    statement
```

```
    /* Counting loops */
```

```
/* Do loop for i = 2, 3, 4, 5, 6, 7 */
for(i = 2; i & 8; i++)
    s = s + i;
```

```
/* Do loop for i = 2, 4, 6 */
for(i = 2; i & 8; i = i + 2)
    s = s + i;
```

```
/* Do loop for i = 4, 3, 2, 1, 0 */
for( i = 5 ; i--; )
    s = s + i;
/*
note: here there is no re-initialization.
The test also does the re-initialization.
This idiom is common because it results
in faster loops. (Counting down towards
zero eliminates a comparison in the loop.)
*/
```

2.6.6.2. Break and Continue statements

1. The **break** statement causes an immediate exit from a loop (or *switch* statement—see below). (In the case of nested loops, only the innermost loop is exited.)
2. The **continue** statement causes an immediate transfer to the test component of a loop.

2.6.6.2.1. Examples

Suppose we normally want to do a loop 10 times, but sometimes want to exit or restart the loop in the middle:

```
for(i = 1; i &= 10; i++)
{
    .
    .
    if ( /* some condition indicating loop exit */ )
        break;          /* Exit loop immediately */
    .
    .
    if ( /* some condition indicating loop re-start */ )
        continue;      /* Go to loop test immediately */
```

2.6.6. For loop

```
        .  
        .  
    }
```

2.6.6.3. Switch statement

Given:

```
switch (expression) {  
  case constant1:  
    statement_list1  
  case constant2:  
    statement_list2  
    .  
    .  
  default:  
    statement_list_default  
}
```

The **switch** statement performs a multi-way branch. The expression is evaluated and then compared to each of the constants in the case statements. When a match is found, the corresponding *statement_list* and **all other following** *statement_lists* are executed. Only when a *break* statement is found, does control exit the switch statement.

2.6.6.3.1. Example

The function below determines the widest displayed line (taking into account tabs). (The source code file [pwidth.c](#) [./src/pwidth.c] is available.)

```
/*  
  print_width finds widest line to be printed  
  when array of characters is displayed  
*/  
#define TABSTOP    (8)  
print_width(s)  
char s[];    /* s an array of characters */  
{  
  int    width, column, i;  
  
  width = column = 0;  
  for( i = 0; s[i] != '\0'; i++) {  
    if ( column > width) width = column;  
    switch( s[i] ) {
```

```

case '\n':    /* newline */
case '\r':    /* carriage return */
case '\f':    /* form feed */
/* Reset column count to zero for new line */
    column = 0;
    break;
case '\b':    /* backspace */
/* Decrement column count for backspace, but
don't make column count negative! */
    if (column > 0 ) column--;
    break;
case '\t':    /* tab */
/* Increment column count to next tab position */
    column = column + TABSTOP - column%TABSTOP;
    break;
default:
/* For regular characters, just increment column count */
    column++;
}
}
return (width);
}

```

2.7. Bit Operators

1. Besides arithmetic and pointer operators, C provides operators on bit patterns.

Table 2.4. Bitwise operators

Operator	Meaning
&	bitwise and
	bitwise or
^	bitwise exclusive or
<<	shift left
>>	shift right
~	invert each bit
	bitwise or

2.7.1. Examples

```
/* Determine number of bits in an integer */
int i;
j = 0;
for(i = 1; i != 0; i = i<<1)
    j++;
printf("ints take %d bits\n", j);

/* Make n = lower 3 bits of i */
n = 7&i;
```

```
/* Invert all bits except least significant 4 bits */
n = (~0xf)^n;
```

1. Note that in the last example we use `~0xf` to indicate a bit pattern of 1's except in the last four positions instead of `0xffffffff0`. The reason is not to save typing; rather, the `~0xf` method is independent of the number of bits in an integer. With the other method, we would have to use `0xffff0` for 16-bit machines.

2.8. Expanded Assignment Statements

To save typing and possibly increase efficiency, the extended assignment operators can be used:

Table 2.5. Assignment operators

Operator
<code>+=</code>
<code>-=</code>
<code>*=</code>
<code>/=</code>
<code>%=</code>
<code><<=</code>
<code>>>=</code>
<code>&=</code>
<code> =</code>
<code>^=</code>

For these operators,

```
x op= y
```

means exactly the same thing as:

```
x = x op y
```

2.9. Conditional (?) operator

1. An expression using the ternary conditional operator has the following syntax:

```
expression1 ? expression2 : expression3
```

The entire expression takes on the value of *expression2* if *expression1* is true or of *expression3* otherwise.

2. For example:

```
x = (i > j) ? i : j;
```

is equivalent to:

```
if ( i > j )
    x = i;
else
    x = j;
```

3. Earlier, we saw another use of the ? operator in the program `up3.c`:

```
putchar(islower(ch) ? ch + 'A' - 'a' : ch);
```

2.10. The Comma (,) operator

1. The comma operator separates expressions into a larger expression; the value of the larger expression is the value of the last comma-separated expression.
2. The comma operator is often used in *for* statements to get the effect of two loop variables.
3. For example:

```
for (i = 0, j = 0; i < 10 ; i++, j += 2)
    /* loop statements */
```

2.10. The Comma (,) operator

effectively defines two loop variables, i and j where i takes the values 0, 1, 2 ... 9 and j the values 0, 2, 4 ... 18.

Chapter 3. Project management and make

3.1. Using separate source code files

When a program consists of more than one function, it is often useful to split the program amongst several *source code* files. Each file contains the code for one or more functions and we call each file a *module*. The advantages to this approach include:

1. Individual modules are smaller, easier to maintain, and faster to compile than complete programs contained in a single file.
2. If modules are written so that they contain a useful, generic set of functions, the modules can be shared by different programs.
3. In particular, the compilation process is actually divided into 2 (or more) separate phases: **1)** *compilation* of source code to object code, and **2)** *linking* of object code files into an executable file.
4. For example, suppose a program is made up of three source code modules *a.c*, *b.c*, and *c.c*. In the first phase, each source module is compiled individually into its corresponding object module. In this case, we would obtain the object code files: *a.o*, *b.o*, and *c.o*. Next, the linker joins the object modules together to produce the executable *ex_name* (where the “*ex_name*” of the executable is defined by the programmer).
5. Suppose that the program is then modified and that the changes only occur in source module *b.c*. There is no need to re-generate the object modules *a.o* and *c.o*; only *b.o* need be re-generated.
6. By using the *make* program maintenance package, the entire program can be re-generated automatically after individual modules have been modified with the minimum possible amount of computer time. We will examine the details of the *make* utility later. Note that `{\em make}` is available for almost all operating systems.
7. We will encourage this style of programming in the seminar; note, however, that the substantial advantages of this method only become truly evident with large projects (several thousand lines of code).

3.2. Scope of variable names

Before looking at how a program is organized into modules, it is important to clarify the *scope* of names in C and how some of the features can be used to implement, at some level, the advantages of pure block structured languages

Locals:

- Variables declared within the body of a function (or any compound statement) are local to the function;
- Locals may not be used by other functions;
- Locals do not retain their values from one call to another.
- Locals are allocated on the stack when the function is entered and de-allocated upon exit.
- They have undefined initial values.

Globals:

- Globals are declared outside of all functions;
- Globals may be accessed by any function;
- Globals are allocated at absolute memory locations;
- Globals are declared outside of all functions;
- Many compilers/linkers initialize globals as zero if not explicitly defined in the declaration; however, you should not rely on this behavior.

Static Locals:

- Like automatic locals for scope rules, but retain their value from one call to another;
- Allocated in absolute memory (like globals).

Static Globals:

- Like normal globals but can only be used within the same source code file.
- Useful for a module whose functions share a a *module* global which is private to the module (i.e. inaccessible by users).

3.2.1. Notes

Some useful rules of thumb:

- Use static locals sparingly.
- When a global seems necessary, prefer static globals within a module (for both function names and variables) over globals. This is reduce the *name space pollution* of global variables.

3.3.1. A Simple Example

- Carefully consider the pros and cons of having global variables vs. parameter passing between functions. (In a nutshell, globals are often a more efficient and “easy solution” for sharing data between related functions; however, passing parameters may be safer! (Functions that manipulate pure or static globals or even static locals are also *not* re-entrant. This may lead to serious problems in the server portion of client-server applications.)

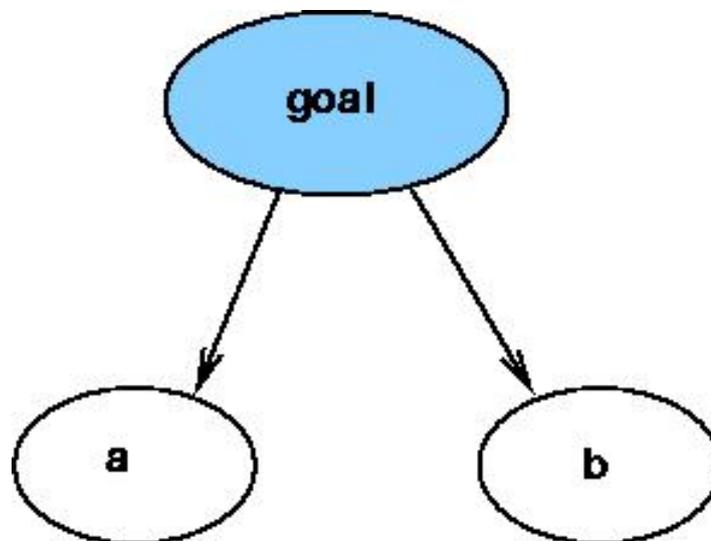
3.3. Make

1. The *make* utility is a useful program maintenance tool for projects split into several modules (i.e. source code files.)
2. The *make* utility automates the task of regenerating *target* files from files that they *depend on*.
3. *Make* uses a data-base (called the *makefile*: usually contained in a file called *Makefile* or *makefile*).
4. *Make* uses the modification time stamps on files to determine if a target should be regenerated. It then invokes the rules found in the *makefile* to do so
5. The most common use of *make* is to recompile a program after editing source code files. *Make* will carry out the minimum number of steps required to keep the various targets up-to-date.

3.3.1. A Simple Example

1. A target called *goal* depends on two files: *a* and *b* as illustrated graphically in the figure below:

Figure 3.1. A simple dependency tree



2. The above dependency tree is described to *make* with the following *Makefile*:

```

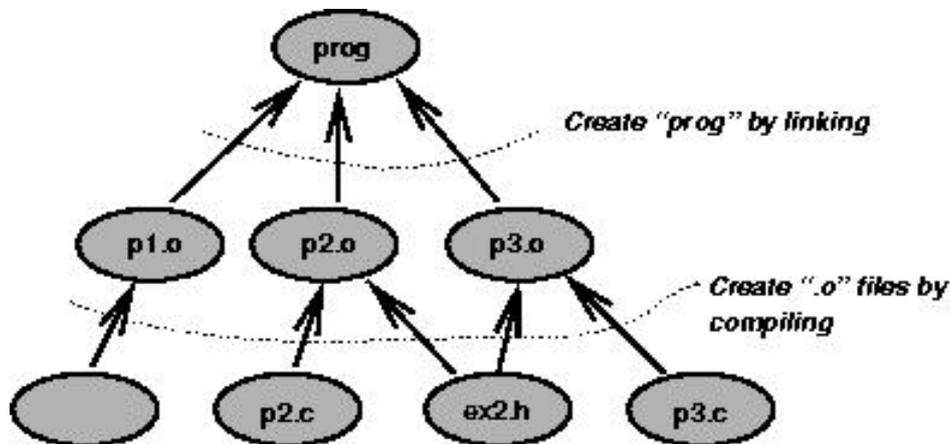
# Simple "Makefile" example
# Anything after a '#' is ignored
# i.e. treated as a comment
goal: a b # Target "goal" depends on "a" and "b"
        touch goal # rule to generate "goal" if "a" or "b"
                # is more recent
# "touch" is a command to set the modification
# time of a file to the current time
b:      # Depends on nothing
        touch b
a:
        touch a
# NOTE: The white space before a rule
# must be TAB.

```

3.3.2. Compilation example

1. A more realistic example of using make involves compiling. The figure below shows the dependencies of a final program, called *prog*, on three object files: *p1.o*, *p2.o* and *p3.o*. If *prog* does not exist or is older than any of the three object files it depends on, it will be regenerated.

Figure 3.2. Dependency tree of simple makefile



2. Similarly, each of the object files depends on a corresponding C source code file. If any of the source code files is newer than its object file, the object file will be regenerated. Since this will make at least one object file newer than the executable, *make* will automatically re-link the object files as well.
3. Finally, note that two source files—*p2.c* and *p3.c*—include the custom header *ex2.h*. Consequently, the object files *p2.o* and *p3.o* both depend implicitly on *ex2.h*. Since *make* detects

3.3.3. Additional remarks

if a target file is older than anything it depends on, the object files `p2.o` and `p3.o` will be regenerated if `ex2.h` is more recent.

4. The Makefile below shows how all of this is described.

```
# Simple "Makefile" example using compilation
prog: p1.o p2.o p3.o
    gcc -o prog p1.o p2.o p3.o
p1.o: p1.c
    gcc -c p1.c

p2.o: p2.c
    gcc -c p2.c

p3.o: p3.c
    gcc -c p3.c

# The following lines give additional
# files that object code depends on;
# in this case, these are header files
# included in the source code.
# If any of these are newer than the
# corresponding object file, the
# rules defined above will be invoked
# to re-create the object code.
p2.o: ex2.h
p3.o: ex2.h
```

3.3.3. Additional remarks

1. Make allows string variables to be assigned and referenced as follows:

```
VAR_NAME = blah blah
.
$(VAR_NAME)
```

2. By default, the first target is made when make is invoked. However, other targets may also be specified. (The next example shows the common targets `clean` and `depend`.)
3. Normally, if a command in a rule has a non-zero exit status, make stops. However, preceding the command with a dash (-) causes make to ignore the exit status.
4. This feature is used in the `clean` target. The rule specifies that all generated files should be deleted. Obviously, if any of these files do not exist, we still want the make program to proceed.

5. Putting these ideas together, we obtain a more elegant Makefile for the previous example:

```
# Simple "Makefile" example using compilation
SRCS = p1.c p2.c p3.c
OBJS = p1.o p2.o p3.o

prog: $(OBJS)
    gcc -o prog $(OBJS)
p1.o: p1.c
    gcc -c p1.c
p2.o: p2.c
    gcc -c p2.c

p3.o: p3.c
    gcc -c p3.c

depend: $(SRCS)
    makedepend $(SRCS)
clean:
    -rm -f $(OBJS) a.out core prog
# DO NOT DELETE THIS LINE - - make depend depends on it.
p2.o: ex2.h /usr/include/stdio.h
p3.o: ex2.h /usr/include/stdio.h
```

3.3.4. Using generic rules

1. You can specify, in general, how to obtain a target from a dependent when they only differ in their suffixes. For example, the C compiler is used to obtain an object (.o) file from a corresponding source (.c) file.
2. The following generic rule states this:

```
.c.o:
    $(CC) -c $*.c
```

where \$(CC) is the name of the system C compiler and \$* is the variable representing the base name of the target.

3. Other builtin make variables include: \$@—the name of the current target; \$<—the name of the dependency file; \$?—the list of dependencies newer than the target.
4. Putting these rules together, we obtain the following Makefile:

```
# Simple "Makefile" example using compilation
SRCS = p1.c p2.c p3.c
OBJS = p1.o p2.o p3.o
```

3.3.5. Parting remarks

```
PROG = prog
HOME = /home/eccles1/kclowes
INSTALL_PATH=$(HOME)/bin
CFLAGS = -c
CC = gcc
.c.o:
    $(CC) $(CFLAGS) *.c
$(PROG): $(OBJS)
    gcc -o $(PROG) $(OBJS)
depend: $(SRCS)
    makedepend $(SRCS)

clean:
    -rm -f $(OBJS) a.out core $(PROG)
install: $(PROG)
    cp $(PROG) $(INSTALL_PATH)/$(PROG)
# DO NOT DELETE THIS LINE - - make depend depends on it.
p2.o: ex2.h /usr/include/stdio.h
p3.o: ex2.h /usr/include/stdio.h
```

3.3.5. Parting remarks

- The `-n` option causes *make* to report the commands it would invoke without actually doing them.
- The `-f file` option allows you to specify a file other than *Makefile* or *makefile*.
- Note that a shell is invoked for each line in a *makefile*'s rule. For example, the two lines:

```
cd sub
ls
```

would invoke a separate subshell for each command; consequently, the `ls` command would execute on the current directory, not the one changed to in the previous sub-shell.

This can be avoided by putting both commands on a single line:

```
cd sub; ls
```

In this case the entire line has its own sub-shell.

Chapter 4. Basic Arrays and Pointers

4.1. Arrays

1. Single dimension arrays of any data type are declared as:

```
char    buffer[120], line[80];
int     freq[30];
double  coeff[N];
```

2. An array dimensioned n has elements indexed from 0 to $n-1$.
3. Note that variably-dimensioned arrays are not allowed in C. Hence, in the above example declaration `double coeff[N]`, N must be a symbolic constant previously defined with a `#define` preprocessor directive.
4. Multi-dimensional arrays are declared as “arrays of arrays”:

```
unsigned short table[20][30];
float    pressure[100][100][100];
double   velocity[100][100][100][3];
```

5. We shall see later that the use of pointers is often more efficient than the use of multiply-dimensioned arrays. (In particular, there are advantages to using an array of pointers to singly-dimensioned arrays instead of doubly-dimensioned arrays.)
6. Note that multi-dimensioned arrays can require a lot of memory. For example, the *velocity* vector defined at each point in a 3-dimensional space with a grid size of 100 in each direction requires $3 \times 100 \times 100 \times 100 \times \text{sizeof}(\text{double}) = 24$ Megabytes of storage!

4.2. Pointers

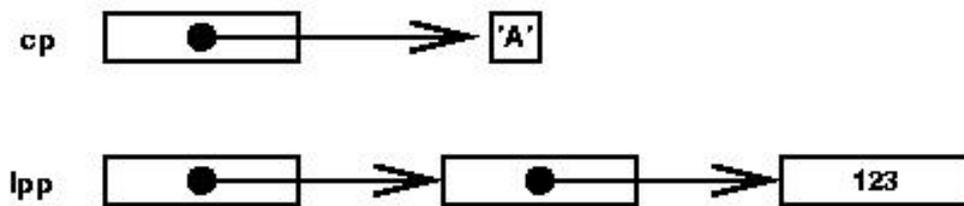
1. A *pointer variable* contains the address of a data object. They are declared as follows:

```
char    *cp;
int     *integerp, **ipp;
```

2. In the above declarations, *cp* is a pointer; **cp* refers to the object that *cp* is pointing to—i.e. a character.
3. The variable *ipp* is a *pointer to a pointer to an integer*. The diagrams below illustrate the relationship of a pointer variable (itself stored somewhere in memory) and the data it points to.

First, suppose *cp* points to the character 'A' stored somewhere in memory and *ipp* points to a pointer pointing to the integer 123 stored somewhere in memory:

Figure 4.1. Pointer visualization



4.2.1. The & Operator

The & operator is used to obtain the address of an object. Thus pointers can be initialized as follows:

```
int i, j, *ip;
.
.
ip = &i; /* Make ip point to i */
j = j + *ip; /* same as j = j + i */
```

4.2.2. Pointer Arithmetic

1. Integers may be added or subtracted from pointers.
2. If n is added to a pointer p pointing to an object obj , p is incremented to point n objects further. (i.e. the numerical value of p becomes $p + n * \text{sizeof}(obj)$).
3. It is possible to perform ordinary arithmetic on pointers (i.e. arithmetic such that adding 1 to the pointer results in incrementing the numerical value of the pointer by 1) by *casting* the pointer to an integer. (More precisely, it should be cast to a *(void *)* or *(size_t)*. These conventions are associated with the *stdlib*—we will examine their precise meaning later.)
4. More precisely, it is normally cast to an unsigned integer, or, a void pointer (which points to data of size 1).

4.2.2. Pointer Arithmetic

- For example, suppose *ip* is 0x1000 (i.e. it points to data at address 0x1000). If the data it points to is an integer and integers occupy 4 bytes, then *ip++* will result in increasing the numerical value of *ip* to 0x1004 (the address of the next integer).
- However, if we use:

```
(unsigned) ip++;  
        /* or */  
(void *) ip++;
```

the numerical value of *ip* will increase from 0x1000 to 0x1001, just like an ordinary number.

4.2.2.1. Strings are character pointers

- A String constant (enclosed in double quotes) are set up as a NULL terminated array of characters in memory.
- The length of the array is the number of characters in the string plus 1 (for the null).
- The *value* of the string constant is a pointer to the first character in the array.
- For example, given the code:

```
char *cp;  
cp = "Hello";
```

the compiler stores the characters for the letters 'H', 'e', 'l', 'l', and 'o' followed by a null terminator (a literal zero) in memory and assigns the variable *cp* the address where the letter 'H' is stored.

The figure below illustrates this:

Figure 4.2. Pointer visualization char *cp = "Hello";)



4.2.2.1.1. Example

Given:

```
char *cp1, *cp2;  
cp1 = "Hello"  
/* Make cp2 point to dynamically allocated memory */
```

```

cp2 = malloc(strlen(cp1) + 1);
/* copy string1 to string 2 */
while (*cp1 != '\0') {
    *cp2 = *cp1;
    cp1++ ; cp2++;
}
*cp2 = '\0';

```

- In the above program fragment we make use of two library routines: `malloc()` and `strlen()`.
- `strlen()` returns the number of characters in a string (not including the null terminator); it is passed the starting address of the string
- Hence `strlen(cp1) + 1` is the total number of bytes required to store the string pointed to by `cp1` (including the null terminator).
- `malloc(n)` dynamically allocates n bytes of memory and returns an address to the first byte allocated.
- Hence, `cp2 = malloc(strlen(cp1) + 1)` makes `cp2` point to the first byte of a dynamically allocated region big enough to hold the string that `cp1` points to.

 The program could have been written more efficiently as:

```

while ( *cp1 != '\0' )
    *cp2++ = *cp1++;
*cp2 = '\0';

```

Finally, the program could have been written even more efficiently as:

```

while (*cp2++ = *cp1++)
    ;

```

Note that there is no loop body! Everything is done in the `while` test including the copy of the null terminator.

4.3. Arrays and Pointers

1. There is a strong relationship between arrays and pointers. In particular, the name of an array is defined in C as being the starting address of the array. This address can then be assigned to a pointer variable, and we can step through the array by changing the pointer instead of using indices. Indeed, this is how arrays are actually implemented internally.
2. Consider the following:

4.4. Functions

```
char c1, c[100], *cp;
int i;

/* The symbol "c" is a pointer to the start of array */
/* Hence, the following is legal */
cp = c;
/* It is equivalent to: */
cp = &c[0];
/* The following are all equivalent: */
c1 = c[i];
c1 = *(cp + i);
/* Curiously, the following are also legal */
c1 = i[c];
c1 = "abcdef"[i];
```

3. The last idiom is quite useful; for example, consider:

```
vowel = "aeiou"[vowel_number];
    /* or */
hex_ascii_code = "0123456789ABCDEF"[i & 0xF];
    /* The above gives the ascii code for the
    hex digit formed by the lower 4 bits of i */
```

4.4. Functions

1. Functions can return elementary data types or pointers to them.
2. Unless otherwise declared, functions are assumed to return integers.
3. Functions returning a data type other than integer must be declared before use.
4. Most compilers allow functions to be declared as type void if they return nothing.
5. Functions arguments are copied onto the stack. The copies are discarded when the function returns. Hence, all arguments are *pass by value*.
6. The same effect as pass by reference can be achieved by passing a pointer to the data.
7. Functions should be declared before they are used; in ANSI C, we can declare a function with a *prototype* that describes the type of its arguments. The compiler can use the prototype to issue warnings about incorrect usage and to generate more efficient code.

4.4.1. Examples

```

/* Prototype for function swap */
void swap(int *, int *);
/* Prototype for function pow */
double pow( double, int);

/* Actual function declaration begins here */
void swap(int * ip1, int * ip2) /* function returns nothing */
{
    int temp;
    temp = *ip1;    /* temp = what ip1 is pointing at */
    *ip1 = *ip2;
    *ip2 = temp;
}
double pow(double x, int i)    /* function returns double */
{
    double temp;
    temp = 1;
    if ( i > 0)
        while(i - - )
            temp = temp*x;
    else
        i = -i;
        while(i - - )
            temp = temp/x;
    return temp;
}

```

4.4.2. Differences between ANSI and K&R C

1. The most visible changes between ANSI and K&R C are in the way that functions are declared.
2. In ANSI C, we use:

```

double pow(double x, int i)
{
    /* body of function */
}

```

3. In K&R C, we use:

4.5.1. Comamnd line arguments

```
double pow(x, i)
double x;
int i;
{
    /* body of function */
}
```

4. Even if you only use ANSI C, you need to be familiar with the old-fashioned method since so much code was written with it. (ANSI compilers do understand the older method.)

4.5. Examples of pointers

The following programs illustrate the elementary use of pointers in very typical applications. In particular, we examine:

- How command line arguments are passed to executable programs.
- How to write a function with a variable number of arguments

4.5.1. Comamnd line arguments

1. When a program is compiled and turned into an executable file, one may wish to alter the behavior of the executable at the time it is invoked by passing it arguments on the command line to invoke it.
2. The simplest such standard command is **echo** (in both Windows and UNIX) which simply echoes its command line arguments to *stdout* when it is invoked.
3. For example:

```
OS_prompt> echo Display this
```

results in the output:

```
Display this
```

4. The command line arguments are passed to the `main()` function of any program in a standard way. In particular:
 - The first argument passed to `main()` is the total number of command line arguments (including the command name). By convention (not a rule that must be obeyed), this argument is called `argc` (“argument count”).
 - The second argument passed is the starting address of an array. By convention is is called `argv` (“argument values”) and is declared as: `char *argv[]`. There are `argc` elements in the array and each element in the array is a pointer to a null terminated string corresponding to

a command line argument. Array element zero (i.e. `argv[0]`) is the command name while array elements 1 to `argc-1` are the command line arguments following the command name.

 In UNIX, it is possible to have different names for the same file. Hence, `argv[0]`—i.e. the particular name that was used to invoke the executable file—can be used to determine what action to take.

For example, the common UNIX commands **mv** (move or rename), **cp** (copy), and **ln** (link) all refer to the identical executable file. By looking at `argv[0]`, the program determines what it should do.

5. The following example shows the code for a modified **echo** command:

```
/*
   myecho echos its command name and
   arguments to stdout.
*/
main(int argc, char *argv[])
{
    int i;

    printf("My argument count is: %d\n", argc);
    printf("I was invoked under the name of %s\n",
           argv[0]);
    printf("My arguments were:\n");
    for (i = 1; i < argc; i++)
        printf("Arg #%d: %s\n",
              i, argv[i]);
    exit(0);
}
```

6. A more interesting example is shown below:

```
#include <stdio.h>
char progame[128];
main(int argc, char * argv[])
{
    float   princ, interest;
    int     i, n_years;

    strcpy(progame, argv[0]);
    if (argc != 4) {
        printf("Usage: %s principal interest num_years\n",
              progame);
    }
}
```

4.5.2. Variable Number of Arguments

```
    exit(1);
}
princ = atof(argv[1]);
printf("argv[1]: %s, princ: %f\n", argv[1], princ);
interest = atof(argv[2]);
printf("argv[2]: %s, interest: %f\n", argv[2], interest);
n_years = atoi(argv[3]);
printf("argv[3]: %s, n_years: %d\n", argv[3], n_years);
interest = 1. + interest/100.;
for(i = 1; i <= n_years; i++) {
    princ = princ*interest;
    printf("After %d years, new principal is: %8.2f\n",
        i, princ);
}
exit(0);
}
```

- The command line arguments are always passed as strings; if these strings represent numbers, they must be converted to the proper format by the programmer.
- The functions `atof()` and `atoi()` (from standard library) perform the conversions.
- Note also how `argv[0]` is copied into the global variable `progname`. Since `argv[0]` is the name under which the program was invoked, this allows generic error handlers to print out the program name since all functions will have access to the global variable `progname`. Note that under Windows, however, `argv[0]` also includes the full path.)

4.5.2. Variable Number of Arguments

1. Functions like `printf()` are passed a variable number of arguments.
2. The number of arguments must, however, be implicit in the arguments themselves.
3. In the case of `printf()`, for example, the number of arguments is equal to 1 plus the number of (non-escaped) `%` characters in the formatting string which must be the first argument.
4. The **cat** example below concatenates any number of strings. The number of strings to be concatenated is explicitly given as the first argument. (Another common way to pass a variable number of arguments is to make the last argument a null pointer.)
5. We will first write this function in a *non-standard* way, illustrating the method most C compilers use to pass parameters to functions. This can be useful information when linking with other languages that use a different convention. It is also a good exercise in the use of pointers.

Warning

However, it will not work on all machines, especially RISC machines which pass the first few arguments in registers, not on the stack. For portability, `stdarg.h` must be used. (We shall see how to use it shortly.)

6. In this example, we assume that arguments are passed to functions by pushing them onto the stack, that the stack grows towards lower memory, and that the first argument is the last one pushed on.
7. The code follows:

```
/*
 * Cat concatenates a variable number of strings, allocates
 * memory for the resulting string and returns a pointer to it.
 *
 * arguments:
 *   n - number of strings to concatenate
 *   args - "n" pointers to NULL terminated character strings
 *
 * returns:
 *   a pointer to the string of the concatenated
 *   arguments
 *
 */

char * cat(int n, char * args)
{
    register int i, len = 0;
    char *malloc();
    register char **s, *x, *start;
    s = &args; /* s points to first argument on stack */

    /* Add up the lengths of all the strings */
    for(i = n; i-- > 0; )
        len += strlen(*s++);

    /* Allocate enough memory for all strings (plus NULL) */
    x = malloc(len + 1);
    start = x;

    s = &args; /* Point back to first argument on stack */

    /* Concatenate all the input strings together into x */
```

4.5.2. Variable Number of Arguments

```
while (n-- ) {
    while (*x++ = *(*s)++); /* Cat next string to x */
    s++; /* Point down stack to next argument */
    x-- ; /* Backspace over NULL */
}

return start;
}
```

4.5.2.1. stdarg() right way to handle variable number of arguments

1. The approach above to a variable number of arguments is useful in showing how parameter passing on the stack is *usually* done and how C can be exploited to take advantage of it.
2. However, while parameters are usually passed on the stack as indicated there, it is not always done this way. (e.g. RISC machines sometimes use registers and the HP 3000 has a stack that grows towards higher memory.);
3. The portable solution is to use `stdarg.h` (`varargs.h` in K&R C compilers).
4. Cat can be re-written more portably as shown below:

```
/*
 *          PORTABLE VERSION using stdarg.h
 *
 */

#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif

char * cat_va(int n, ...)
{
    register int i, len = 0;
    char *malloc();
    va_list ap; /* points to each arg in turn */
    register char *x, *start;

    va_start(ap, n); /* Make ap point to first anonymous arg. */

    /* Add up the lengths of all the strings */
    for(i = n; i-- ;)
```

```
    len += strlen(va_arg(ap, char *));

    /* Allocate enough memory for all strings (plus NULL) */
    x = malloc(len + 1);
    start = x;
    va_end(ap);

    va_start(ap, n);    /* Make ap point to first anonymous arg. */

    /* Concatenate all the input strings together into x */
    while (n - - ) {
        /* Cat next string to x */
        while (*x++ = (*(char **)ap)++);
        va_arg(ap, char *);
        x-- ; /* Backspace over NULL */
    }

    va_end(ap);
    return start;
}
```

Chapter 5. Pointers to functions

1. Another kind of pointer used in C is a *pointer to a function*. This is a very powerful technique that we will examine in greater detail later on. For the moment, we will concentrate on how such pointers are declared and used; specifically, we will look at their use in the standard library sorting function `qsort()`.
2. Pointers to functions contain the address of a function. At the machine level, this means that the value of a function pointer is the address of the first executable instruction of the function.
3. Function pointers can be passed to other functions which can then use the passed function.
4. Very common use: standard library sort function. The sort function must be passed a function pointer of a routine that can compare two elements of the type to be sorted. (Hence the sort utility can be used to sort any kind of data.)
5. The ANSI standard library describes a function `qsort()` (quicker sort) that will sort an array of objects “in place”.
6. Sorting “in place” means that the `qsort` routine gets the array as a passed parameter and, when it returns, the array is modified so that its elements are sorted. Furthermore, no auxiliary arrays are created.
7. The algorithm is not trivial; however, if it were implemented directly, it would require different implementations for sorting arrays of chars, ints, doubles, etc.
8. Examination of the routine, however, indicates that the only differences between sorting arrays of one data type or another are related to the ways in which the elements are compared.
9. The solution to the problem of different versions of the sort function for arrays of different data types can be resolved by allowing the programmer to pass, as a parameter, a pointer to a function that will compare elements of the array.
10. This is the solution used in the standard library `qsort()` function.
11. The function prototype for the `qsort` function is:

```
void qsort((char *) base,  
          size_t num,  
          size_t width,  
          int (*compare) (const void *elmt1, const void *elmt2));
```

The meanings of the arguments are:

base:

The starting address of the array to be sorted.

num:

The number of elements in the array.

width:

The size (in bytes) of each element of the array.

compare:

A pointer to a function that compares two data items and returns a negative number if the first is less than the second, 0 if they are equal and a positive number if the first is greater than the second.

12.



- Note the keyword `const` in the prototype for the `compare()` function.
- This means that the value of the object pointed to will not be modified by the function `qsort`. It does not mean that the corresponding value in the calling routine is a constant.

13. To invoke `qsort()` to sort an array of 100 integers, use:

```
#include <stdlib.h>

{
    int i_array[100];
    .
    qsort(i_array, 100, sizeof(i_array), icomp);
    .
}

int icomp(const int *a, const int *b)
{
    if ( *a > *b)
        return 1;
    else if (*a > *b)
        return -1;
    return 0;
}

/* NOTE: the following version of icomp() would be better in this case
 *      (although the above illustrates the more generally
 *      applicable pattern.)
 *
 * int icomp(const int *a, const int *b;
 * {
 *     return (*a - *b);
 * }
```

```
*/
```

5.1. Complex declarations

1. We have now seen all the elementary data types C can handle.
2. Declarations for complex variables that, for example, are functions pointers that return the base address of an array of integers can be difficult to declare.
3. Consider the following examples:

```
/* funcp is a pointer to a function (that
 * has a single parameter of type double)
 * that returns the address of an array of N
 * integers
 */
int (*funcp(double))[N];

/*
 * aupp is a matrix of pointers to pointers
 * to structures
 */
struct {
    int i;
    int j;
} **aupp[M][N];
```


Chapter 6. Parsing

6.1. Compilers: An Overview

1. In an abstract sense, a computer program is simply a sequence of bytes (or bits).
2. From a computer's point of view, more concretely, this is exactly what the source code for a high level language (HLL) is.
3. The source code is stored in some file on a disk and the job of the compiler is to read this file and then somehow transform it into another file containing machine language that will perform the precise tasks defined by the semantics (i.e. what a human would understand it to mean) of the HLL source code file.
4. When the compiler, which is merely another program, “reads” the source code file it simply brings this file, one byte at a time, into its memory.
5. This reading of the source code is entirely different from our own idea of what reading is all about; thus if two bytes—0x69 and 0x66— are read in sequence there are no hardware bells to signal that these correspond to the ASCII representation for the letters ‘i’ and ‘f’ and that their sequential occurrence introduces an `if` statement.
6. When a human reads, there is little conscious effort required in translating weird black shapes into “letters”, grouping the letters into “words” and analyzing the groupings of words by their context into “meanings”.
7. Even with all these advantages, however, humans can find the process of translation difficult. The translation process—i.e. going back from abstract “meanings” to “groups of words” (in another language!) to words to letters and finally to “funny black shapes” (which may be quite different from the original if we are going from English to Arabic)—requires conscious effort from the human.
8. But when we want a computer to compile a program, we are asking the computer to perform an exact translation between two languages (the C language to machine language for example) and, at the same time, depriving the computer of all the intuitive knowledge of “reading”, “meaning”, “context”, etc. that we all have.
9. To write a compiler, not only do we have to describe in the utmost detail the translation of “meaning” (or semantics) into a different language; we must also describe (i.e. write a program) how to interpret sequences of bytes into words and how to group words into semantic units.
10. This looks like a big job!
11. If one simply attacks this big job as yet another programming assignment, it does indeed take a lot of effort. The first Fortran compiler back in the 1950's, for example, was written from this perspective and it has been noted that that:

The first Fortran compiler, for example, took 18 staff-years to implement.²

—Aho

12. Today, however, the compilation process is much better understood. We now have a theory of compilation and it is not unreasonable to expect an undergraduate student to write a complete compiler as a course project.
13. With this theory, we divide the compilation process into three major steps:
 - a. lexical analysis
 - b. parsing
 - c. code generation.

Each of these stages are described briefly in the following sections.

6.1.1. Lexical analysis

1. Lexical analysis is used to translate the incoming sequence of bytes read from the source code file into more meaningful *tokens*.
2. The lexical analyzer can recognize sequences of characters as words.
3. For example, input that we read as:

```
thing := stuff*10;
```

would be read by the lexical analyzer as the following sequence of 18 bytes:

```
74 68 69 6e 67 20 3a 3d 20  
73 74 75 66 66 2a 31 30 3b
```

4. The lexical analyzer translates these 18 bytes into a sequence of 6 tokens as follows:

```
<identifier (with value of ``thing'')>  
<assignment operator>  
<identifier (with value of ``stuff'')>  
<multiply operator>  
<constant (with value of 5)>  
<statement terminator>
```

² Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, page 2.

6.1.2. Parsing

5. This is not too complicated to do. The language definition requires that keywords be separated by white space (blanks, tabs or newlines) or by special characters (like = ; > etc. not allowed to be part of keywords or identifiers (i.e. symbols).
6. Hence the lexical analyzer reads one byte at a time and collects them until it hits white space or a special character.
7. It has then collected a “word” and compares it against its list of keywords.
8. If it finds a match it outputs a *token* representing that keyword; if no match is found an “identifier token” is output along with some link to the token's “value”. The lexical analyzer also recognizes that a word beginning with a digit should be output as a “constant token” with a link to the actual digit string.
9. Finally, special characters (like ; + *) or groups of special characters (like := >=) are also recognized and have their own individual tokens.
10. The number of types of tokens is quite limited. This is a marked contrast from natural languages like English with vocabularies that are bigger by many orders of magnitude.
11. This limited vocabulary of programming languages (also called *formal languages* in contrast to *natural languages*) makes the lexical analyzer's job much easier.
12. Note that the lexical analyzer makes no judgements about the significance or sensibleness of the stream of tokens its generates.
13. (However, the lexical analyzer may be able to detect a non-sensical token such as a string like 123ax5 which, because it starts with a digit should generate a constant token while the string as a whole is not a number.)

6.1.2. Parsing

1. Once the lexical analyzer has converted the source code into a token stream, the *parser* (or semantic analyzer) takes over.
2. Its job is to find meaningful patterns in the token sequence and to pass these “meanings” to the final stage of compilation—the code generator— which produces the final machine language translation of the original source code.
3. Finding meaningful patterns is greatly simplified by the formal definition of the source code language grammar.
4. The syntax (or grammar) of a formal language like C can be completely described using the Backus-Nauf Form (BNF) notation. This notation describes how tokens can be combined to form semantic units.

5. Some of these semantic units are very simple. The “addop” semantic unit, for example, is either an addition (+) or subtraction (-) operator and BNF uses the following notation to describe this:

```
<addop> ::=
    <+ operator> | <- operator>
```

6. The '|' (vertical bar) symbol is used to describe alternatives. Note that the '+' and '-' characters are simple tokens recognized by the lexical analyzer.
7. The semantic analyzer or parser uses the above rule or pattern to recognize these simple tokens as a more abstract semantic unit.
8. The BNF description also describes how the addop unit can be combined with other semantic units to form yet more complex abstract entities.
9. For example, we know that we can add or subtract two or more “terms” and that a programmer specifies this by placing '+' or '-' characters between the terms.
10. Let's call something like “a + b - c” an expression. The general BNF description for an expression is:

```
<expression> ::=
    <term> {<addop> <term>}
```

11. The '{' and '}' (curly braces) indicate that the symbols within them may be repeated zero or more times.
12. Assuming for the moment that a term is a simple variable, all the following are valid expressions according to the above BNF definition:

```
<term> and 0 repetitions of <addop> <term>:
a
<term> and 1 repetition of <addop> <term>:
c - d
<term> and 2 repetitions of <addop> <term>:
a + b - c
```

13. To use the BNF rule in finding these expressions, the parser would first recognize the initial term.

6.1.2. Parsing

14. It would then determine if the next semantic entity was an `<addop>`: if it was not, the parser would return the simple expression; otherwise it would look for the `<term>` that must follow the `<addop>` and then repeat the sequence looking for another `<addop>` or the end of the expression.
15. The ultimate goal of the parser is to combine all the tokens and all the semantic units into the highest-level unit of all: the `<program>`. As the higher-level units are formed, information about their structure is passed to the code generator which then has sufficient information to perform “actions” based on the semantic structure.
16. These actions generate the machine code equivalent to the semantic structures of the source code.
17. We will consider the the example of an *assignment statement* to illustrate this point.

Consider the BNF:

```
<assignment statement> ::=
    <variable> := <expression>

<expression> ::=
    <term> { <addop> <term> }

<addop> ::= + | -

<term> ::=
    <factor> { * <factor> }

<factor> ::=
    <variable> | ( <expression> )
```

18. If we look at how expressions are defined here, we see that any arithmetic expression involving the '+', '-' and '*' operators and the '(' and ')' grouping characters is defined.
19. The definitions are recursive as an expression is defined as a grouping of `<term>`s; terms in turn are defined using `<factor>`s; finally factors are defined using `<variable>`s and again with `<expression>`s.
20. This method of defining something in terms of itself looks strange at first glance but it is a powerful method for simple, elegant definitions of grammars.
21. Consider how the parser would treat the following expression:

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

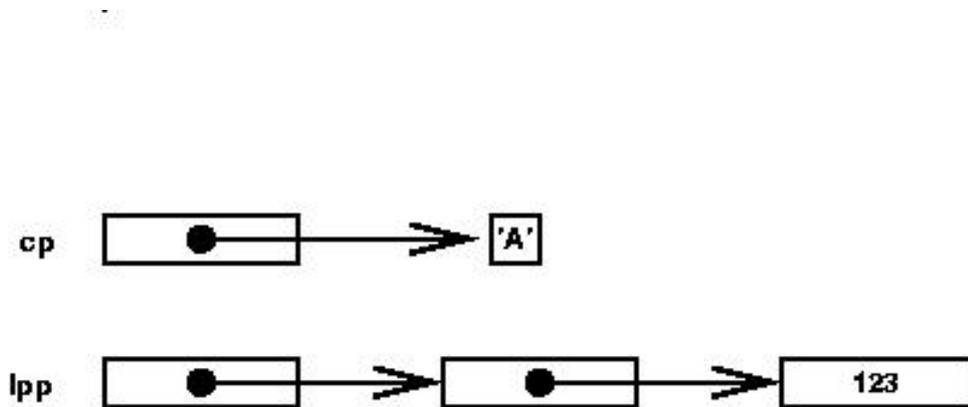
22. When the expression detecting routine is called, it first invokes another routine to look for the first thing in an expression—in this case a term. As soon as the term routine is called, it immediately looks for its first item—a factor.
23. A factor, in turn, has two alternative definitions: either it is a simple variable or it is an expression enclosed within parenthesis.
24. The factor routine examines the first character to see if it is a simple variable or a '('.
25. In this case, it finds the '(' special character; it then moves to the next input token which it expects to be the start of an expression and calls the expression recognition routine once again.
26. When the expression routine starts a second time, it again calls term and then factor.
27. This time, however, factor finds a simple variable and returns immediately. The term routine then looks for a '*' to see if it should call factor again; it doesn't find one, so it returns immediately. Now the second invocation of expression looks for a '+' sign to see if it should again call term again to find out what should be added to the first term it found.
28. Expression finds a simple variable, 'c', on the other sign of the + sign. It then looks for another '+' or '-' to see if it should continue. Since the next character is a ')', this second invocation of expression terminates and returns to the factor routine that called it.
29. Factor then expects to find a ')' to balance the first one it detected and it returns to term after finding the next token following the ')'. Term examines this token to see if it is a '*'. Since it is, term calls factor again to find the other factor of the multiplication. Term then finds no more '*'s and returns to the first invocation of expression.
30. We are now almost done. Expression looks for another '+' or '-', finds none, and terminates. The entire expression has now been completely parsed. We now know what has to be added, subtracted and multiplied and are in a position to generate machine code to perform these operations and evaluate the expression.
31. We can, of course, write a program to parse expressions based on the above syntax. Such a program, written in C, is shown below. This program does more, however, than merely recognize expressions. It also does some translation; in particular, it translates algebraic notation for expressions defined in the BNF for expressions into *Reverse Polish Notation* (RPN).
32. With Reverse Polish Notation (used for example to enter expressions in Hewlett-Packard calculators), the expression is read and each variable or constant is pushed onto a stack. When an operator is encountered, the top two elements are popped off the stack, the operation performed on them, and the result pushed back on the stack.
33. Consider the following expression and its RPN equivalent:

6.1.2. Parsing

```
(a+b)*(c + d +e) //algebraic
ab+cd+e+* //RPN equivalent
```

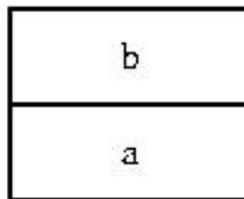
34. If the RPN expression is examined one token at a time, then, just before encountering the first '+' operator, the stack would look like the figure below:

Figure 6.1. RPN stack before processing first '+' in ab+cd+e+*



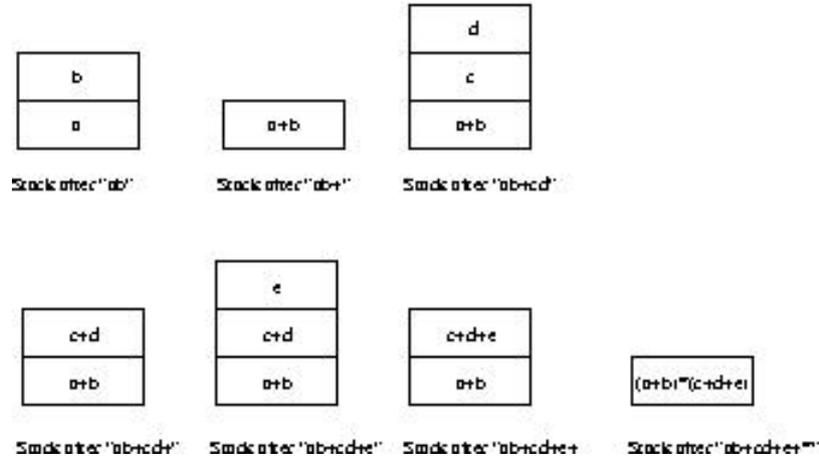
35. After processing the first two characters (i.e. “ab”) in the RPN expression, the stack would look like the figure below:

Figure 6.2. RPN stack after processing “ab” in ab+cd+e+*



After processing
ab+cd+e+*

36. It would then undergo the following transformations:

Figure 6.3. RPN stack when interpreting $ab+cd+e+*$ 

37. To make the parser program convert the algebraic expression to reverse polish notation at the same time that it is parsing it we simply add the following: print identifiers as they are encountered but defer printing a '+', '-' or '*' until both of its terms (or factors) have been found.
38. The C program (available at [parseRPN.c](#) [./src/parseRPN.c]) is:

```
#include <stdio.h>
/*
 * The program works on expressions conforming to the following BNF syntax
 * rules:
 * <expression> ::= <term> [ <addop> <term> ]
 * <term> ::= <factor> [ <factor> ]
 * <factor> ::= ( <expression> ) | <id>
 * <addop> ::= + | -
 * <id> ::= any single character except ()*+-
 *
 * How to run the program: The program reads algebraic expressions for stdin and
 * writes their Reverse Polish Notation (rpn) equivalent to stdout. In order
 * to avoid seeing all of this on the screen, redirect stdin or stdout. For
 * example, postfix1 < exprs will read expressions from the file "exprs"
 * (which you have to create previously).
 */

char    ch;

main()
{
    find();
    do
```

```
{
    expression();
    putchar('\n');
} while (ch != '.');
}

find() /* Read input until non-white space or
        * end-of-line reached */
{
    while (((ch = getchar()) == ' ') || (ch == '\n'))
        ;
}

expression()
/*
 * Look for an expression and print
 * the + or - operand if and when a "term"
 * is found after such an operand
 */
{
    char    op;
    term();
    while ((ch == '+') || (ch == '-'))
    {
        op = ch;
        find();
        term();
        putchar(op);
    }
}

term()
/*
 * Look for a term and print the * operand if and when a "factor" is found
 * after such an operand
 */
{
    factor(); /* Find the initial factor and print rpn for
               * it */
    /*
     * Move around the "factor*factor" loop as long as the next character
     * is the '*' operand'

```

```

    */
while (ch == '*')
{
    find();    /* Move to next non-blank character */
    factor(); /* Find the factor after the '*' and,
               * if it is a simple identifier, print it */
    putchar('*'); /* Complete the rpn for the factor by
                  * printing the '*' operand */
}
}

factor()
    /* Look for a factor; if the factor is a simple identifier, print it */
{
    if (ch == '(')
    {
        find();    /* Move to the next non-white character */
        expression(); /* Find the expression & print rpn for it */
        /* After returning "ch" should be a ')' if syntax is correct */
    } else
    {
        /*
         * If not an expression enclosed in "...", it has to be a
         * simple identifier so print it
         */
        putchar(ch);
    }
    find();    /* Now move to next non-blank character */
}

```

6.1.2.1. Parsing C declarations

1. The syntax of C declarations (especially pointers to functions!) can be confusing. We now look at the formal BNF specification of declarations and how we can write a program to interpret these declarations.
2. These notes are based on the programs on pages 122–126 of *The C Programming Language* by Kernighan and Ritchie (2nd edition).
3. First, the simplified BNF syntax of a declaration is:

6.1.2. Parsing

```
<dcl> ::= {*} <direct_dcl>

<direct_dcl> ::= <name>
                | '(' <dcl> ')'
                | <direct_dcl> '(' ' ' ')'
                | <direct_dcl> '[' <size> '']'
```

4. A C declaration can be interpreted by a relatively simple recursive-descent parser of the type we have looked at.
5. The source code (available at [parse_dcl.c](#) [./src/parse_dcl.c]) is:

```
/**
 NAME
   parse_dcl
 PURPOSE
   This program parses C declaratations (reading from stdin)
   and outputs an English language description of each declaration's
   meaning to stdout.
 NOTES
   This program is a lsightly different version of the program
   given in Section 5.12 (pages 122--126) of the K & R book,
   second edition.

   A simplified form of C declaration is used, as described
   in the following BNF:

   <typed_declaration> ::= <type> <declaration>
   <declaration> ::= { '*' } <direct_declaration>
   <direct_declaration> ::= <name>
                           | '(' <declaration> ')'
                           | <declaration> '(' ' ' ')'
                           | <declaration> '[' <dimension> ']'
   <name> ::= a sequence of characters
   <dimension> ::= { <digit> }

 EXAMPLES
   Assuming the program is called parse_c_decs, then
   parse_c_decs << XXXX
       char *** abc;
       char *simple();
       char (*foo)();
       char (*( *bar())[ ]());
```

```

int >(*foobar[2][3])();
float (*f())[5]();
XXXX

```

produces:

abc is a pointer to pointer to pointer to char

simple is a function returning pointer to char

foo is a pointer to function returning char

bar is a function returning pointer to array[] of pointer to function returning char

foobar is a array[2] of array[3] of pointer to function returning pointer to function returning int

f is a pointer to function returning pointer to array[5] of function returning float

BUGS

This is demonstration program only. It does not understand the full C syntax of declarations.

For examples, structs are not supported, type qualifiers like "const" or "volatile" are not supported, function argument types are not supported, etc.

HISTORY

kclowes (Ken Clowes) - Sept 25, 1993: Created.

```

***/
#include <stdlib.h>
#include <strings.h>
#include <stdio.h>
#include <ctype.h>

#define MAX_TOKEN_LENGTH 100
#define MAX_DESCRIPTION_LENGTH 1000

enum
{

```

```
NAME,  
BOTH_PARENS,  
BRACKETS,  
LEFT_PARENS,  
RIGHT_PARENS,  
DECLARATION_END  
};  
  
typedef struct  
{  
    int type;  
    char value[MAX_DESCRIPTION_LENGTH];  
} token_t;  
  
static token_t token; /* next token from input stream */  
static char description[MAX_DESCRIPTION_LENGTH];  
static char name[MAX_TOKEN_LENGTH];  
static int line;  
static int n_errors;  
  
void declaration(void);  
int gettoken(void);  
void direct_declaration(void);  
  
int main(int argc, char *argv[])  
{  
    char data_type[MAX_TOKEN_LENGTH];  
  
    for (line = 1, n_errors = 0; gettoken() != EOF; line++)  
    {  
        /* 1st token is data declaration type */  
        strcpy(data_type, token.value);  
        description[0] = '\0';  
        declaration();  
        if (token.type != DECLARATION_END)  
        {  
            fprintf(stderr, "Syntax error on line %d\n", line);  
            n_errors++;  
        }  
        printf("%s is a %s %s\n", name, description, data_type);  
    }  
    exit(n_errors);  
}
```

```
}

/**
NAME:
    int gettoken()
PURPOSE:
    Reads the next token from stdin and set the global variable
    token to it.
ARGS:
    > None
RETURNS:
    < The token type: i.e. NAME, BOTH_PARENS, BRACKETS, LEFT_PAREN,
        RIGHT_PARENS, or the last return value form getchar
NOTES:
    The "last return value" from getchar may, or course, be EOF.
**/

int gettoken()
{
    int ch;
    char *p = token.value;

    while((ch = getchar()) == ' ' || ch == '\t' || ch == '\n')
        ;
    if (ch == '(')
    {
        if ((ch = getchar()) == ')')
        {
            strcpy(token.value, "()");
            return token.type = BOTH_PARENS;
        }
        else
        {
            ungetc(ch, stdin);
            return token.type = LEFT_PARENS;
        }
    }
    else if (ch == '[')
    {
        for(*p++ = ch; (*p++ = getchar()) != ']');
        ;
        *p = '\0';
        return token.type = BRACKETS;
    }
}
```

```
else if (isalpha(ch))
{
    for (*p++ = ch; isalnum(ch = getchar()); )
        *p++ = ch;
    *p = '\\0';
    ungetc(ch, stdin);
    return token.type = NAME;
}
else if (ch == ';')
    return token.type = DECLARATION_END;
else if (ch == ')')
    return token.type = RIGHT_PARENS;
else
    return token.type = ch;
}

/**
NAME:
    declaration
PURPOSE:
    Parses the following BNF.
    <declaration> ::= { '*' } <direct_declaration>

ARGS:
    > None.
RETURNS:
    < Nothing.
NOTES:

**/

void declaration(void)
{
    int num_stars;

    for(num_stars = 0; gettoken() == '*'; num_stars++)
        ;
    direct_declaration();
    while(num_stars-- > 0)
        strcat(description, " pointer to");
}

/**
NAME:
```

```

direct_declaration
PURPOSE:
  Implements the parsing of:
    <direct_declaration> ::= <name>
                                | '(' <declaration> ')'
                                | <declaration> '(' ')'
                                | <declaration> '[' <dimension> ']'

ARGS:
  > None
RETURNS:
  < Nothing
NOTES:
  Called from declaration, so leading <declaration>, if any, already
  parsed.
  **/

void direct_declaration(void)
{
  if (token.type == NAME)
    strcpy(name, token.value);
  else
    if (token.type == LEFT_PARENS)
      {
        declaration();
        if (token.type != RIGHT_PARENS)
          {
            fprintf(stderr, "missing ) on line %d\n", line);
            n_errors++;
          }
      }
    else
      {
        fprintf(stderr, "expected name or (dcl) on line %d\n", line);
        n_errors++;
      }
  for(gettoken();
      token.type == BOTH_PARENS || token.type == BRACKETS;
      gettoken())
    if (token.type == BOTH_PARENS)
      strcat(description, " function returning");
    else
      {
        strcat(description, " array");
        strcat(description, token.value);
      }
}

```

6.1.2. Parsing

```
        strcat(description, " of");  
    }  
}
```


Chapter 7. Data Structures

7.1. Structures and Unions

1. A *struct* (like a “record” in Pascal) groups data items into a single logical unit. For example:

- ```
/*
The following declares a structure data type
called "complx".
No storage is reserved; only the name and data
type of the component fields of the structure
are defined.
*/
struct complx {
 double real;
 double im;
};
```

- ```
/*
The following declares two arrays of 100 items each.
Each array item is a structure of the type
defined above
*/
struct complx z[100], y[100];
```

2. Portions of a structure are accessed as follows:

```
z[2].real = y[1].real * y[1].real - y[1].im * y[1].im;
```

7.2. Structure Examples

1.

```
struct person {
    char *first_name;
    char *last_name;
    char sex;
    int age;
} persons[10];

/*
In the above declaration, both the definition
```

```

of the structure fields and the allocation of
of an array of 10 such structures is combined
into a single declaration.
*/

persons[0].last_name = "Jones";
persons[0].first_name = "Robert";
persons[0].sex = 'M';
persons[0].age = 23;

```

- Note that structures can be assigned. Thus:

```
persons[4] = persons[8];
```

will copy structure #8 to structure #4.



Beginners are sometimes surprised that structures can be assigned but not arrays. They become even more amazed when told that if they make the array the only member of a structure, it can be assigned

Can you see why C works this way?

7.3. Using typedef

- It is often convenient to use a “typedef” to give a user defined name to a declaration. (This is much like the “type” keyword in Pascal.)
- For example, to define a new data type called “Complex” that corresponds to the complex number data structure we saw earlier, we write:

```
typedef struct complx {
    double real;
    double im;
} Complx;
```

- The word Complx will now be interpreted by the compiler as the name for a data type. Thus:

```
Complx z[100], y[100];
```

- Note: typedefs are useful for things other than structures. For example, consider:

```
typedef unsigned char byte;

byte var1, var2;
```

This creates a new data type called `byte` which is an alias for an unsigned char. Variables `var1` and `var2` are then declared as type `byte`.

7.4. Pointers to structures

1. Pointers to structures are extremely useful for common data structures such as linked lists or trees.
2. A linked list is a group of structures where each structure contains a pointer to the next structure in the list. The declaration of such a structure is an example of self-referential data structures.
3. For example:

```
struct node {
    int data;
    struct node * next;
};
```

4. Note that if a typedef is used, the typedef name cannot (unfortunately) be used in the structure declaration. Thus we must write:

```
typedef struct node {
    int data;
    struct node * next;
} node, * nodePtr;
node all_nodes[NN];
nodePtr node_ptr = &all_nodes[3];
```

5. However, this blemish can be avoided by separating the typedef from the struct declaration as follows:

```
typedef struct node node, * nodePtr;
struct node {
    int data;
    nodePtr next;
};
node all_nodes[NN];
nodePtr node_ptr = &all_nodes[3];
```

7.5. Single Linked List Example

The following code fragments illustrate the methods used in allocating storage for new structures, accessing a specific structure on a linked list and stepping through a linked list.

```
typedef struct elem elem, * elemPtr;
struct elem {
    char    name[30];
    elemPtr next;
};
elemPtr first, temp;
/* i.e. first is a pointer to a structure called elem;
The structure contains two fields:
a 30 character array and a pointer called next
to an other structure of the same type */

first = (elemPtr) malloc(sizeof elem);
strcpy(first->name, "Smith");
temp = first->next
      = (elemPtr) malloc(sizeof elem);
temp->next = (elemPtr) malloc(sizeof elem);
.
/* Print out 3rd name */
printf("%s\n", first->next->next->name);
.
/* Step through list */
for(temp = first; temp != NULL ; temp = temp->next)
    printf("%s\n", temp->name);
```

7.6. Initializing Linked lists in declarations

1. The following code fragment shows how a linked list can be set up at compile time.
2. The example below sets up a circular linked list of three nodes.
3. Note that structures on the list referred to that have not yet been declared require an extern declaration before they are referred to.
4. The sample code is:

```
typedef struct node  node, * nodePtr;
struct node {
    int data;
```

```
    nodePtr next;
};
extern node node2;
extern node node3;

node node1 = {
    1,
    &node2
};

node node2 = {
    22,
    &node3
};

node node3 = {
    333,
    &node1
};
```

7.7. Unions

1. A *union* is a data item that can hold different types of data (at different times).
2. For example, we may wish to manipulate a number that may be stored as either an integer or as a floating point number. The following union allows this:

```
union number {
    int integer;
    float floating_point;
} n1, n2;
```

3. The compiler will force the size of number to be big enough to hold the largest variant. It is the programmer's responsibility to keep track of how the data is represented within the union.
4. The union member is referenced much like structure elements. For example, to print n1 (stored as an integer) and n2 (stored as a floating point number), we use:

```
printf("n1: %d; n2: %f\n", n1.integer, n2.floating_point);
```

5. Unions are often combined with structures, with one field in the structure indicating the format of the union.
6. For example, suppose we sometimes wish to store complex point numbers in rectangular notation and sometimes in polar coordinates. We could use the following:

```
struct cartesian {
    float real;
    float imag;
};

struct polar {
    float mag;
    float angle;
};

typedef struct {
    enum { CARTESIAN, POLAR } complex_type;
    union {
        struct cartesian cart;
        struct polar polar;
    } value;
} Complex;
```

7.8. Example: Doubly Linked List

1. A “doubly-linked list” has both forward and backwards pointers; this kind of list requires more storage (two pointers per structure) and more time to insert or delete an item. Many problems are more easily solved with the added flexibility, however. The following examples show how doubly linked lists can be manipulated.
2. These examples also illustrate:
 - Splitting a problem into smaller parts;
 - Using function pointers as arguments.
 - Using the comma operator to obtain 2 loop variables in a *for* loop. (This is a common idiom when manipulating linked lists.)
3. The following sections give details.

7.8.2. Main routine

7.8.1. Header file `doubleLinkedList.h`

The header file defines the basic data structures and macros. The source code (available at [doubleLinkedList.h](#) [`./src/doubleLinkedList.h`]) is:

```
/* Header file describing structures
   for a doubly-linked list */
#include <stdlib.h>
typedef struct rec {
    char word[80];
    struct rec *next;
    struct rec *prev;
} REC, *RECP;

#define NILREC ( (RECP) NULL)
#define allocrec (RECP) malloc(sizeof(REC))
```

7.8.2. Main routine

```
/* Copyright K. Clowes (kclowes@ee.ryerson.ca) 1996
 * May be copied, modified, etc. by anyone.
 * The Copyright notice does NOT have to be retained.
 */

/*
   This routine reads words from stdin
   and inserts them in sorted order into
   a doubly linked list.

   It then prints out the list in forward
   and reverse order.
 */

#include <stdio.h>
#include "double.h"
RecPtr headPtr, tailPtr;
RecPtr Forward();
RecPtr Backward();

main()
{
    /* Initialize the list as empty */
    headPtr = tailPtr = NILREC;
```

```

    /* Create the doubly linked list */
    BuildList();

    /* Output the list in forward and reverse order */
    PrintList(headPtr, Forward);
    printf("\n");
    PrintList(tailPtr, Backward);
}
\end{verbatim}
\subsection{Build list}

%\verbatimlisting{c_progs/doubleLinkedLists/dbuild.c}
\begin{verbatim}
/* Copyright K. Clowes (kclowes@ee.ryerson.ca) 1996
 * May be copied, modified, etc. by anyone.
 * The Copyright notice does NOT have to be retained.
 */

/*
BuildList reads stdin a line at a time
and builds a sorted doubly-linked list
from the input.
*/
#include <stdio.h>
#include "double.h"
extern RecPtr headPtr, tailPtr;
BuildList()
{
    RecPtr newPtr, prevPtr, currentPtr;

    while((newPtr = allocrec)
        && (scanf("%s", newPtr->word) != EOF)) {
        for(currentPtr = headPtr, prevPtr = NILREC;
            currentPtr != NILREC && strcmp(currentPtr->word, newPtr->word) < 0;
            prevPtr = currentPtr, currentPtr = currentPtr->next);

        newPtr->next = currentPtr;
        newPtr->prev = prevPtr;
        if ( currentPtr == NILREC)
            tailPtr = newPtr;
        else
            currentPtr->prev = newPtr;
        if ( prevPtr == NILREC)

```

7.8.3. Print list

```
    headPtr = newPtr;
    else
        prevPtr->next = newPtr;
    }

    if ( newPtr == NILREC) {
        fprintf(stderr, "!!!!!! OUT OF MEMORY !!!!!!");
        exit(1);
    }
}
```

7.8.3. Print list

```
/* Copyright K. Clowes (kclowes@ee.ryerson.ca) 1996
 * May be copied, modified, etc. by anyone.
 * The Copyright notice does NOT have to be retained.
 */

/*
PrintList prints the contents of a linked list
using a supplied function to determine the
next element on the list.

Arguments:
RecPtr start: ptr to the first item on the list

getnext: pointer to a function to return the next
item from the current item.
*/

#include <stdio.h>
#include "double.h"

PrintList(RecPtr startPtr, RecPtr (*getNextProc)(RecPtr))
{
    RecPtr tempPtr;

    for(tempPtr = startPtr;
        tempPtr != NILREC;
        tempPtr = (*getNextProc)(tempPtr))
        printf("%s\n", tempPtr->word);
}
```

```
RecPtr Forward(RecPtr recPtr)
{
    return(recPtr->next);
}

RecPtr Backward(RecPtr recPtr)
{
    return(recPtr->prev);
}
```

7.8.4. Notes

If the linked list were implemented as a circular list, insertion and deletion would be simpler. The circular list would be initialized with a structure containing a word lexically beyond any possible real words (e.g. “zzzzz”).

Chapter 8. The Preprocessor

1. The C preprocessor is the first stage of the compilation process. It transforms the original source code into a “transformed” source code that is fed to the actual compiler.
2. Note that the preprocessor is, in effect, a text manipulator (or a batch editor). Both its input and output are text files. It is quite possible to use the C preprocessor for languages other than C.
3. To use the C preprocessor for other languages requires either a special option to the C compiler or is a stand-alone program.
4. The concept of automatic translation of one form of source code to another has been extended in projects such as D. Knuth's *web* project. Here, a single “source code file” is transformed into separate documentation and implementation files. Knuth's original work was targeted to Pascal, but his ideas have been ported to other languages (such as the *cweb* package for C).

8.1. Using the Preprocessor

1. Preprocessor commands begin with a # and we have already used two such commands: #include and #define.
2. Proper use of the pre-processor helps in making programs easier to read (and write!), easier to maintain, more portable, and more efficient. We now examine the features of the preprocessor in more detail.
3. We will first look at those preprocessor features common to both ANSI and K&R C. Later, we will look at the enhanced capabilities only available in ANSI C.

8.1.1. #include

1. The #include directive simply inserts the named file into the source code. The file is called a “header file” and, by convention, its name ends with .h.
2. The most common type of include statement looks like:

```
#include <header_name.h>
```
3. This is usually used to include a header file associated with functions (or macros) supplied with the compiler. For example, #include <stdio.h>.
4. The compiler knows where to find the file.
 - In UNIX, it usually looks for the header file in /usr/include and /usr/local/include.
 - Under Windows using the **gcc** compiler and the **cygwin** package, it looks in C:\cygwin\usr\include.

5. However, the compiler can be instructed to look for the include file in other directories by giving their names to the `-I` option in the compiler. Thus, if the compiler should look for header files in directory `/usr/me/my_includes`, use the command line:

```
UNIX_PROMPT% cc -I/usr/me/includes source_file.c
```

6. The other way to specify an include file is with:

```
#include "path_file"
```

7. In this case, the exact path (relative or absolute) for the file is given.

8. For example, if the header file `head.h` is in the same directory as the source code, simply use:

```
#include "head.h"
```

9. It is also very common for a large project to be divided into several sub-directories: `src` for source code, `doc` for documentation, `include` for header files, etc.

10. Thus a source code files in sub-directory `src` can access header files by moving up to the parent directory (`..`) and down to the `include` directory as follows:

```
#include "../include/head.h"
```

8.1.2. #define

1. We have used `#define` to define the values of symbolic constants. It can be used to define anything.
2. Symbolic constants can be re-used in other `#define` statements as shown below:

```
/*Clock freq (kHz) divided by prescaler*/  
#define CLOCK 4000/32  
#define INT_TIME 200 /* in units of milliseconds */  
#define COUNT (INT_TIME*CLOCK)  
/* Split 24 bit COUNT into 8 bit parts (hi, mid, lo) */  
#define COUNTHI (COUNT/(256*256))  
#define COUNTMID (COUNT/256 - COUNTHI*256)  
#define COUNTLO (COUNT%256)
```

3. `#define` directives can also be *parameterized* to define *macros*. For example:

8.1.3. #ifdef and #ifndef

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

4. If, later in the source code, we see:

```
j = max(i+2, k);
```

the preprocessor replaces this with:

```
j = ((i+2) > (k) ? (i+2) : (k))
```

5. A macro appears to behave like a function. However, it is more efficient since it is expanded in-line. (There is no subroutine call or return.)
6. For example, we have been using `getchar()` as if it were a function. In fact it is (usually) a macro defined in `stdio.h`.
7. There are dangers, however, in the fact that macros and functions look the same. In particular:
- You cannot take the address of a macro (and pass it as function pointer).
 - You must be careful about any possible side effects associated with a macro. For example, consider:

```
m = max(++i, ++j);
```

where `max()` is the parameterized macro previously defined.

It will be expanded to:

```
m = (((++i) > (++j) ? (++i) : (++j))
```

As a result, either `i` or `j` will be incremented twice!

8.1.3. #ifdef and #ifndef

1. The `#ifdef` (*if defined*) preprocessor directive is used to conditionally compile certain sections of code. This often leads to more portable code and code that can easily be recompiled to behave in a different, configurable way.
2. The form:

```
#ifdef Symbol  
    ...  
    ...  
#endif
```

results in the statements between the `#ifdef` and the matching `#endif` being compiled only if the string `Symbol` has been defined by a previous `#define` statement (or command line option when the compiler was invoked).

3. For example, consider:

```
#ifdef MOT68K
typedef unsigned int big_numbers;
#endif
#ifdef INTEL8088
typedef unsigned long big_numbers;
#endif
```

4. Here, the proper `typedef` will be compiled so that the type `big_numbers` will be an unsigned 32-bit integer on both PCs (which use Intel 8088 chips) and UNIX 680x0 machines.
5. A very common use for `#ifdef` is to surround debugging statements with `#ifdef DEBUG` as shown below:

```
#ifdef DEBUG
    printf("i: %d now\n", i);
#endif
```

6. When developing the code, `DEBUG` is defined; hence the debug statements are compiled and executed at run time. Once the program works, it is re-compiled without defining `DEBUG` and it will now run without the debugging statements.
7. Note that this is preferable to deleting the debugging statements. After all, a hidden bug may be reported later on. In this case, the program has to be debugged again but there is no need to re-write the debugging statements; simply re-compile with `DEBUG` defined.
8. Note that a symbol can be defined on the command line that invokes the compiler instead of in the program itself. Thus, to compile with debugging statements, we write:

```
SHELL_PROMPT% gcc -DDEBUG source_file.c
```

9. The directive `#ifndef` works just like `#ifdef` except that the statements following it up to the `#endif` are only compiled if the symbol is *not* defined.

8.1.4. #undef

1. The effect of a `#define` statement can be undone with the `#undef` directive.

8.1.5. ANSI C preprocessor

2. This is often useful if you want to use a function for something that is normally a macro. For example, if you want to use your own function `getchar()` (which is a macro expansion by default), use:

```
#undef getchar
.
.
int getchar() {
    /* Your function body */
}
```

3. Another way of forcing `getchar()` to be a function rather than a macro is to use: `(getchar)()`.

8.1.5. ANSI C preprocessor

1. In addition to the familiar `#include`, `#define`, `#ifdef`, etc., the following features have been added to the preprocessor in ANSI C.
2. The `#if constant integer expression` has been added for conditional compilation. The “constant integer expression” is considered true if it evaluates non-zero and false if it is zero. The expression can use C style syntax as well as the `defined(name)` which evaluates as true if “name” has been defined.
3. The new directives `#else` and `#elif` (else if) have also been included. For example:

```
#if SYSTEM == UNIXV.3\
    || SYSTEM == SUNOS\
    || SYSTEM == XENIX
#define UNIX
#endif
#if !defined(UNIX) && SYSTEM != MSDOS
#error Do not know operating system
#endif
#if defined(DEBUG)
#if defined(UNIX)
#define INCLUDE unixdebug.h
#elif defined(MSDOS)
#define INCLUDE msdosdebug.h
#endif
#else
#if defined(UNIX)
#define INCLUDE unix.h
#elif defined(MSDOS)
#define INCLUDE msdos.h
#endif
```

```
#endif
#include < HEADER >
```

4. The new operators # and ## can be used to generate strings by concatenation by the preprocessor.
5. When a parameter name in a macro is preceded by a #, the parameter will be quoted and concatenated with any surrounding strings.
6. The ## operator is used to quote and concatenate adjacent macro arguments.
7. The predefined identifiers `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, and `__STDC__` have been added to the preprocessor. They contain, respectively, the decimal number of the current line, the name of the current source code file, the date and the time. `__STDC__` is defined as 1 if the compiler conforms to the ANSI standard. For example:

```
#define err( m )    fprintf(stderr,\
                    "Error on line %d:" #m "\n", __LINE__)
...
    err(print this message after colon);
```

8. The use of ## is shown below:

```
#define instance( prefix, root, postfix)\
    prefix ## root ## postfix
...
i = j + instance(pre, _body_, 5);
/* Above is creates i = j + pre_body_5; */
```

Chapter 9. The Standard library

1. An accomplished C programmer does not re-invent the wheel. The standard library of functions defined by ANSI C includes a good number of useful routines that should not be re-written.
2. For example, some of these routines (especially the string and memory search and copy ones) may be hand-crafted in assembler to exploit the underlying machine architecture to the maximum. This level of detail is something that most of us wish to avoid.

Of course, one has to know what they are. We now examine the most important ones.

9.1. Strings

Use:

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

to use any of these functions.

strlen(char * cp)

returns the number of characters in its string argument;

strcmp(char * source, char * target)

compares (character by character) its two string arguments and returns -1, 0, or 1 if the first is alphabetically before, identical or after the second;

strcat(char * to, char * this)

concatenates its second string argument to its first;

strcpy(char * copy_to, char * copy_from)

copies its second string argument to its first;

1. Many of these have *n*-variants that limit the number of characters to be compared, copied, etc. For example,

```
strncpy(char * to, char * from, int n)
```

copies at most *n* characters from “from” to “to”.

2. These versions are often preferable; for example, to prevent buffer overflows.

9.2. ctype.h

This include file contains several very efficient macros for determining character type. Use of these macros requires the statement:

```
#include <ctype.h>
```

int isdigit(int ch)

Evaluates true if the character is a digit;

```
int isspace(int ch)
```

Evaluates true if the character is a white space character (e.g. tab, space, newline, etc.);

```
int isalpha(int ch)
```

Evaluates true if the character is a letter of the alphabet;

```
int isalnum(int ch)
```

Evaluates true if the character is a letter of the alphabet or a digit;

```
int ispunct(int ch)
```

Evaluates true if the character is a punctuation character;

```
int islower(int ch)
```

Evaluates true if the character is a lower case alphabetic;

```
int isupper(int ch)
```

Evaluates true if the character is a upper case alphabetic;

Other macros convert the character argument into another character:

```
int toupper(int ch)
```

Converts the character to upper case (does nothing if it is already upper case);

```
int tolower(int ch)
```

Converts the character to lower case (does nothing if it is already lower case);

9.2.1. Implementation of ctype.h

1. The `ctype.h` header file is a good example of how the power of the preprocessor can be exploited to yield very efficient code.
2. Here is a typical (abridged) example:

```
#define _U 01
#define _L 02
#define _N 04
#define _S 010
#define _P 020
#define _C 040
#define _B 0100
#define _X 0200
extern char _ctype[];
#define isalpha(c) (( _ctype+1)[c]&(_U|_L))
#define isupper(c) (( _ctype+1)[c]&_U)
#define islower(c) (( _ctype+1)[c]&_L)
#define isdigit(c) (( _ctype+1)[c]&_N)
#define isxdigit(c) (( _ctype+1)[c]&_X)
#define isspace(c) (( _ctype+1)[c]&_S)
#define ispunct(c) (( _ctype+1)[c]&(_P))
```

9.3. stdio.h

```
#define isalnum(c)    ((_ctype+1)[c]&(_U|_L|_N))
#define isprint(c)   ((_ctype+1)[c]&(_P|_U|_L|_N|_B))
#define isgraph(c)   ((_ctype+1)[c]&(_P|_U|_L|_N))
#define iscntrl(c)   ((_ctype+1)[c]&_C)
#define isascii(c)   ((unsigned)(c)<=0177)
#define _toupper(c)  ((c)-'a'+'A')
#define toupper(c)   (islower(c) ? _toupper(c) : (c))
```

3. Note how the macro like `isdigit` works. Given:

```
#define _S    010
#define isspace(c)  ((_ctype+1)[c]&_S)
```

4. The array `_ctype` is 257 characters long and each entry contains a bit pattern indicating if the corresponding ascii code is a lower case letter, a digit, etc. (each of the 8 bit positions corresponds to one of the primitive types).
5. The bitwise “and” (`&`) isolates an appropriate bit of the `_ctype` array element and the expression evaluates to either 0 or 010 (i.e. 8 decimal) which are interpreted respectively as false and true. (Note that the macro exploits the fact that C considers any non-zero value to be true. It is only the results of logical operations (`&&`, `>=`, etc.) that guarantee that “true” will be 1 (one).)

9.3. stdio.h

1. Include

```
#include <stdio>
```

before invoking the very commonly used functions/macros of this package.

2. We have already used some of these such as:

- `printf()`
- `getchar()`
- `putchar()`

as well as constants such as `EOF` (“End Of File”).

3. Another commonly used function is `scanf()` which performs formatted input where the format is specified in a string (like `printf()`). The arguments to `scanf()` must be passed by reference (i.e. pointers to the actual arguments).

Warning

Note that `scanf()` can cause security violations (through buffer overflows). It should be avoided in any sensitive programs (such as SUID code)

9.3.1. File I/O

We also need `stdio.h` to perform file I/O (on other than *stdin* and *stdout*).

9.3.1.1. Opening a file

1. Before being used, files must be opened with the `fopen()` function:

```
#include <stdio.h>

FILE *filep;

if ((filep = fopen("myfile", "r")) == NULL) {
    /* handle error */
}
```

2. The file called “myfile” is opened for reading with the `fopen()` call.
3. `fopen()` returns a pointer to a `FILE` structure (defined in `stdio.h`, its details do not concern us here). If it cannot open the file, `NULL` (also defined in `stdio.h`) is returned.
4. The returned file pointer must be used to access the file later on.
5. In general, the first argument to `fopen()` is the name of the file and the second argument is the *mode*. Legal modes are:
 - **r** for reading a file;
 - **w** for writing to a file from the beginning (and creating it if it does not exist). If it does already exist, the previous contents will be overwritten.
 - **a** for appending (writing) to a file from the end of an existing file. If the file does not already exist, it is created.
 - **r+** for reading and writing to an existing file;
 - **w+** for reading and writing to a file (if the file does not already exist, it is created)
 - **a+** for reading and appending (writing) to a file from the end of an existing file. If the file does not already exist, it is created.

9.3.1.2. Writing to a file

1. The easiest way to write to a file is to use `fprintf()`.
2. It works just like `printf()`, except it has an additional initial argument for the file pointer.
3. For example, to write to file with the handle `file1p` (assumed to be already opened), use:

```
fprintf(file1p, "Hello, world: 1 + 1 = %d\n", 1+1);
```

4. Note that the operating system automatically opens 3 files for you: `stdin`, `stdout` and `stderr`. We have, of course, used the first two extensively. The third should be used for all error messages.

By default, `stderr` is connected to the screen just like `stdout`. However, if `stdout` is re-directed or piped, `stderr` is not. Hence, even with a redirected command, the error messages will still appear on the screen.

5. The following are legal:

```
fprintf(stdout, "Hello, world\n");  
/* In fact, printf is really a macro that expands:  
    printf("Hello, world\n");  
    to the above fprintf statement */  
fprintf(stderr, "Hello, world\n");
```

9.3.1.3. Reading a file

1. The simplest way to read a file is one character at a time is `getc(FILE * stream)`.
2. Note that the `getchar()` function (previously used) is just a simple way to invoke `getc(stdin)`.

9.3.1.4. Closing a file

Once you have finished with a file, close it with:

```
close(file_ptr);
```

9.4. assert.h

1. A sanity check in a program can use the `assert` macro. For example:

```
#include <assert.h>  
assert(i > j);
```

2. Should the condition be false, an error message will be printed to *stderr* indicating the failed condition, the line number and the file name. The program then aborts.
3. If `NDEBUG` is defined, the `assert` macro expands to a null expression.
4. Examining an implementation of the `assert` macro can be instructive. Here's a typical version:

```
/* Typical assert.h header */
#undef assert
#ifdef NDEBUG
    #define assert(test) ((void) 0)
#else
    void _Assert(char *);
    #define _STR(x) _VAL(x)
    #define _VAL(x) #x
    #define assert(test) ((test) ? (void) 0 \
        : _Assert(__FILE__ ":" _STR(__LINE__) " " #test))
#endif
```

9.5. Memory

Standard routines exist for allocating and releasing memory dynamically and or copying, comparing, and searching memory. (Some of these functions are declared in `string.h`, others in `stdlib.h`)

We first consider the memory allocation and release routines. The primary memory allocate routines are:

void * malloc(size_t size):

allocates *size* bytes and returns a pointer to the area or `NULL` if no memory is available.

void * calloc(size_t n_elem, size_t size):

allocates and clears *n_elem* items each of *size* bytes; returns a pointer to the area or `NULL` if no memory available.

void * realloc(void * buf, size_t size):

reallocates the area pointed to by *buf* so that it is *size* bytes long.

The memory block routines are:

void * memmove(void * to, void * from, size_t count):

copies *count* bytes from *from* to *to* **and** ensures that if the areas overlap, the original *from* bytes in the overlapping region are copied before being overwritten; it returns a pointer to *from*.

void * memcpy(void * to, void * from, size_t count):

like `memmove()` but can handle overlapping regions in any way.

9.7. Handling errors

`int memcmp(void * buf1, void * buf2, size_t count):`

compares at most *count* bytes and returns a negative number, zero, or a positive number depending on whether *buf1* is less than, equal to or greater than *buf2*.

9.6. stdarg.h

1. A portion of a typical `stdarg.h` header file is shown below:

```
typedef char *va_list;
#define va_start(ap,v) ap = (va_list)&v + sizeof(v)
#define va_arg(ap,t) ((t *) (ap += sizeof(t)))[-1]
#define va_end(ap) ap = NULL
```

2. Note that the `va_arg(ap, t)` macro does essentially what was done by hand in the first *cat* program we examined previously.
3. The header also illustrates the use of type casts:
 - The `((t *))` casts the entire expression into a pointer of the proper type.
 - The `ap` (“argument pointer”) variable is incremented by the size of this type (hence pointing to the next argument to be retrieved) and the `-1` index retrieves the appropriate address of what `ap` was pointing to before being incremented.

9.7. Handling errors

1. Many of the system calls (the routines documented in Section 2 of the UNIX manual and `int16` calls on DOS machines) return 0 if successful, -1 if an error is detected (and possibly some other values). When an error occurs, the global variable `errno` is set to a number indicating the nature of the malfunction.
2. The include file `errno.h` defines symbolic constants for all the kernel error numbers. A short excerpt from this file (for UNIX) is shown below:

```
#define EPERM 1 /* Not super-user */
#define ENOENT 2 /* No such file or directory */
#define ESRCH 3 /* No such process */
#define EINTR 4 /* interrupted system call */
#define EIO 5 /* I/O error */
```

3. Since no one likes numerical error codes as messages, the function `perror(char * my_msg)` can be used to output the user message `my_msg` to `stderr` followed by an English description of the problem indicated by the global `errno` variable.

4. A useful universal error handler using these features is shown below:

```
#define err_msg(msg) my_perror(msg,  
    " at line: %d in %s: ", __LINE__, __FILE__)  
  
my_perror(char * a1, char * a2, int l, char * f)  
{  
    perror(a1);  
    fprintf(stderr, a2, l, f);  
    fprintf(stderr, "\n");  
}
```

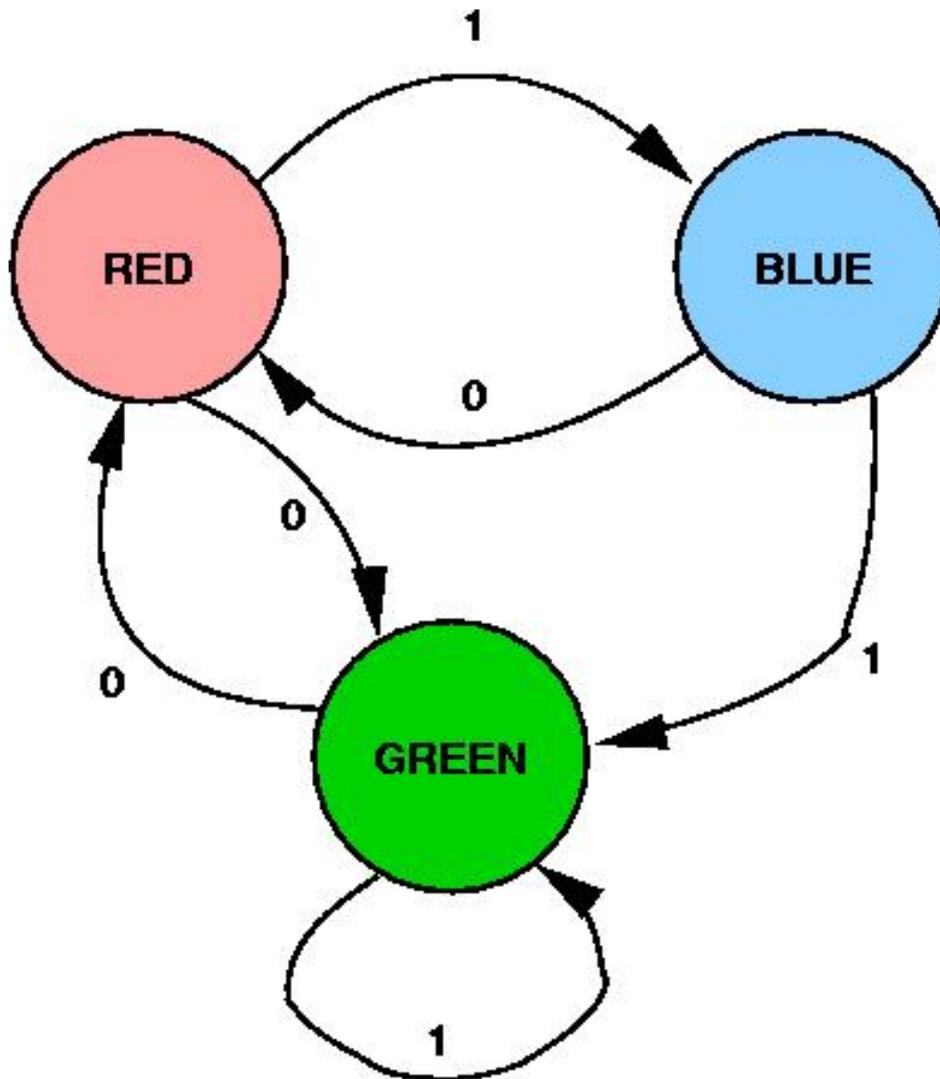
Chapter 10. Data-driven Programming

1. Solutions to problems can be described with algorithms (programs) or data organization (structures).
2. Many problems are best solved with simple algorithms and more complex data structures than the other way around.
3. Programs *driven by data* are much easier to adapt to different situations (only the data need be changed).
4. Often the data itself can be generated with machine aid giving yet an even more flexible solution to the original problem.
5. Sometimes the algorithm required to manipulate the more complex data structure is itself so simple that it is almost “like data” and the program itself can be generated automatically!
6. **THINK** about data-driven solutions before blindly writing a program.

10.1. Example—Finite State Machine

1. Consider the state machine defined by the diagram below:

Figure 10.1. A simple state machine



2. We now examine different ways to implement a C program to simulate this state machine.

10.1.1. The BAD Way

1. The following is the most atrocious C program I hope you will ever see. Hopefully, a bad example will encourage thoughtful solutions.

```
/*  
This program illustrates the WRONG WAY of  
writing programs. The program is an attempt to  
simulate a state machine having three states;  
the algorithm chosen seems straight forward,  
but solves the problem through brute force
```

10.1.1. The BAD Way

instead of examining the fundamentals.

The program "goodstate3" shows the correct method.

```
*/  
  
#include <stdio.h>  
#include <ctype.h>  
#define RED (0)  
#define GREEN (1)  
#define BLUE (2)  
  
main()  
{  
    int    switch_code, c, new_state, state;  
  
    state = RED;  
    printf("Initial state is RED\n");  
    printf("Enter switch_code value: ");  
  
    while ((c = getchar()) != EOF ) {  
        switch_code = c - '0';  
        if (state == RED) {  
            if (switch_code == 0) {  
                new_state = GREEN;  
            } else  
                new_state = BLUE;  
        } else if (state == GREEN) {  
            if (switch_code == 0)  
                new_state = RED;  
            else {  
                new_state = BLUE;  
            }  
        } else if (switch_code == 0)  
            new_state = BLUE;  
        else  
            new_state = RED;  
        printf("New state is ");  
        state = new_state;  
        if (state == RED)  
            printf("RED\n");  
        else if (state == GREEN)  
            printf("GREEN\n");  
        else  
            printf("BLUE\n");  
        printf("\nEnter switch_code value: ");  
    }  
}
```

```

    while (!isdigit(c = getchar()))
        ;
    ungetc(c, stdin);
}
}

```

10.1.2. A BETTER Way

1. Recognizing that the entire pictorial representation of a state machine may be translated into data structures, the far superior solution to the the state machine problem is shown below:

```

/*

This program illustrates the BETTER WAY of writing programs.
We again attempt to simulate a state machine
having three states.

The algorithm used here is simplicity itself (the heart of
the program is one line long and can be used for any state
machine of arbitrary complexity!).

All the information about the state machine is maintained
as data (in data structures) not in the program code itself.
*/

#include <stdio.h>

typedef struct State State, *StatePtr;
/* Note that a state has 3 associated properties:
 *   - the state's name;
 *   - an indicator of the next state if the input is 0
 *   - an indicator of the next state if the input is 1
 *
 * This initial description of the state's fields
 * can be improved.
 *
 * In particular, the last 2 properties (next state if input is 0,
 * next state if input is 1) share a common property: each indicates
 * a "next state".
 * Furthermore, the one to follow is chosen using the input (either
 * 0 or 1).
 *
 * This suggests putting both of these "next state indicators" into
 * an array. The program can use the input to index into the

```

10.1.2. A BETTER Way

```
* array (i.e. either next[0] or next[1]).
*
* With this approach, the program does not have to use
* If/Else logic to select which arrow to follow; if the
* input is "input_value", the next state can be directly
* accessed as next[input_value].
*
* Finally, this approach is much more general and flexible.
* Suppose, for example, that we had to simulate a state machine
* that had 8 possible inputs.
* All that need be done would be to dimension the "next" array
* to have 8 elements and define the data structures properly.
*
*/

struct State {
    char * name;
    StatePtr next[2];
};

State red, green, blue;
/* Define the fields of each state */
State red = {"Red", {&green, &blue}};
State green = {"Green", {&red, &green}};
State blue = {"Blue", {&red, &green}};

int main()
{
    int sw;
    int ch;
    StatePtr currentStatePtr = &red;

    printf("%s\n", currentStatePtr->name);
    while((ch = getchar()) != EOF) {
        if((ch != '0') && (ch != '1')) /* Ignore all chars except 0 & 1 */
            continue;
        sw = ch - '0';

        /* Heart of program! */
        currentStatePtr = currentStatePtr->next[sw];

        /* Display new state */
        printf("%s\n", currentStatePtr->name);
    }
}
```

```
}
```

10.1.3. Doing It All Automatically

1. The only state machine dependencies in the previous code is the declarations and definitions.
2. Let's write a C program that takes a description of a state machine with any number of states and any number of inputs. The program is:

```
/*
genStateMachine takes the description of a state machine (from stdin)
and produces (on stdout) the C source code to simulate
the state machine.
```

The format of the input describing the state machine must be:

```
number of states
number of switches
name of 1st state
name of 2nd state
.
.
name of nth state
next state when in 1st state and switch input = 0
next state when in 1st state and switch input = 1
. . .
. . .
next state when in 1st state and switch input = (2**n_switches) -1
next state when in 2nd state and switch input = 0
. . .
. . .
next state when in 2nd state and switch input = (2**n_switches) -1
. . .
. . .
. . .
next state when in nth state and switch input = (2**n_switches) -1
```

Normally the program would be run with redirection of stdin & stdout.

For example, to generate a C program called test.c from a state machine description in file "test.state", use:

10.1.3. Doing It All Automatically

```
genStateMachine < rgbMachine.state > rgbMachine.c
```

Of course, "cc -o rgbMachine rgbMachine.c" is then used to compile the mechanically generated program; "rgbMachine" is subsequently used to run the simulator.

Note that the format of the input description of the state machine is very strict.

The solution, naturally, is to write yet another program that allows a user to enter the description of a state machine in a much more convenient (i.e. "user-friendly") way and which in turn mechanically generates the more formal description required here.

```
*/

int n_switches, n_states, n_arrows, i, j;
char next_state[16], state_name[16][100];

#include <stdio.h>
main()
{
    scanf("%d%d", &n_states, &n_switches);
    n_arrows = power(2, n_switches);

    printf("/* This program was generated mechanically by genStateMachine */\n");
    printf("#include <stdio.h>\n");
    printf("#include <ctype.h>\n");
    printf("typedef struct State State, *StatePtr;\n");
    printf("struct State {\n");
    printf("    char * name;\n");
    printf("    StatePtr next[%d];\n");\n", n_arrows);
    printf("State ");

    for(i = 0; i < n_states; i++) {
        scanf("%s", state_name[i]);
        printf("%s", state_name[i]);
        printf("%s", i < n_states-1 ? ", " : ";\n");
    }

    for(i = 0; i < n_states; i++) {
        printf("State %s = {\\"%s\", {", state_name[i], state_name[i]);
        for(j = 0; j < n_arrows; j++) {
            scanf("%s", next_state);
            printf("&%s%s", next_state, j < n_arrows-1 ? ", " : "}};\n");
        }
    }
}
```

```

}

printf("int main()\n{\n  int sw;\n  int ch;\n  StatePtr currentStatePtr = &");
printf("%s;\n", state_name[0]);
printf("  printf(\"%%s\\n\", currentStatePtr->name);\n");
printf("  while((ch = getchar()) != EOF) {\n");
printf("    if (!isdigit(ch)) continue;\n");
printf("    sw = ch - '0';\n");
printf("    currentStatePtr = currentStatePtr->next[sw];\n");
printf("    printf(\"%%s\\n\", currentStatePtr->name);\n");
printf("  }\n  exit(0);\n}\n");
}

power(n, i)
  int n, i;
{
  int temp;

  temp = 1;
  while (i > 0) {
    i = i - 1;
    temp = temp*n;
  }
  return temp;
}

```

10.1.3.1. Example

1. To use the **genStateMachine** to generate (for example) the source code for the 3-state “Red-Green-Blue” machine, we first prepare a file (called `rgbMachine.state`) that describes the state machine in the specified format as follows:

```

3
1
red
green
blue
green
blue
red
green
red
green

```

10.1.3. Doing It All Automatically

2. Next, generate the C source code (in the file `rgbMachine.c`) from the description with the command:

```
genStateMachine < rgbMachine.state > rgbMachine.c
```

3. Examine the generated file `rgbMachine.c`. It should be:

```
/* This program was generated mechanically by genStateMachine */
#include <stdio.h>
#include <ctype.h>
typedef struct State State, *StatePtr;
struct State {
    char * name;
    StatePtr next[2];
};
State red, green, blue;
State red = {"red", {&green, &blue}};
State green = {"green", {&red, &green}};
State blue = {"blue", {&red, &green}};
int main()
{
    int sw;
    int ch;
    StatePtr currentStatePtr = &red;
    printf("%s\n", currentStatePtr->name);
    while((ch = getchar()) != EOF) {
        if (!isdigit(ch)) continue;
        sw = ch - '0';
        currentStatePtr = currentStatePtr->next[sw];
        printf("%s\n", currentStatePtr->name);
    }
    exit(0);
}
```

- 4.

10.1.3.2. Notes

1. It would probably be better to use a “scripting language” (such as Perl, python, etc.) for at least part of the program generator.
2. In addition, the unvarying part of the program should be in files instead of embedded in `printf()` statements.

3. One could consider using a formal language as a front end to the program generator. The formal language would describe the state machine; the language could be interpreted by a parser of the type we saw previously.
4. For example, we might want to describe the machine with statements like:

```
STATE_NAMES = RED, GREEN, BLUE, YELLOW
SWITCH_NAME = BUSY, READY
RED to GREEN if 1x
RED to RED if 0x
....
```

10.2. Example—FSM and function pointers

1. The state machine can be made more general by having a pointer to a function defined for each possible transition. Consider, for example, the following declarations of a more complex state machine:

```
/* States for sets */
enum SET_STATE {
    SET_OFF,
    SET_READY,
    SET_BUSY_1,
    SET_HOLD_1,
    SET_BUSY_2,
    SET_HOLD_2,
    N_SET_STATES
};
/* A structure as follows defines
the behavior for a single state */
typedef struct {
    /* Next state to go to */
    int next_state;
    /* Function to implement transition */
    int (*transition_func)();
    /* Secondary function passed to first */
    int (*second_func)();
} TRANSITION_STRUCTURE;

#ifdef GLOBAL_DECLARE
EXTERN TRANSITION_STRUCTURE
    set_state_machine [N_SET_STATES] [N_EVENTS];
#else
EXTERN TRANSITION_STRUCTURE
```

```
        set_state_machine [N_SET_STATES] [N_EVENTS] = {
/* Transitions if in      SET_OFF */
{
/* EVENT_LOGON */      { SET_READY, set_logon, NULL },
/* EVENT_LOGOFF */     { SET_OFF, ignore_event, NULL },
/* EVENT_FORCEOFF */   { SET_OFF, ignore_event, NULL },
/* EVENT_RING_START */ { SET_OFF, ignore_event, NULL },
    ... etc.
/* Do set state machine */
old_set_state = set_state[event.set_num];
new_set_state =
set_state_machine[old_set_state][event_code].next_state;
secondary_func =
    set_state_machine[old_set_state][event_code].second_func;
if ( (*set_state_machine[old_set_state][event_code].transition_func)
    (event, event_code, new_set_state, secondary_func)
    == 0) {
set_state[event.set_num] = new_set_state;
}
}
```

10.3. Menu Driven Code

1. Menu driven applications are often preferred by occasional users.
2. Data-driven techniques the best way to implement user menu interface.
3. Typically, a menu application displays a list of selections on the screen.
4. The user selects an item by typing in the menu number (... or moving the mouse, ... or highlighting items with cursor keys, etc.);
5. The menu-handler program then invokes a specific routine to handle the user's request or signals that the request is invalid.
6. Set up the following array of structures with each structure corresponding to a menu selection PLUS one additional structure for invalid selections (this will be the last one).

Each structure contains the following:

- A pointer to the function to handle the menu item
- A string indicating the name (on the screen) of the menu selection.
- The column and row where the menu item should be displayed.

- A pattern (say a single letter) that the user must enter to select this item
7. A step-by-step algorithm to handle menu selection is:
 1. Read the user's response.
 2. Store the user's response in the pattern part of the structure corresponding to erroneous input (this structure must be the last one in the array).
 3. March through the array of menu item structures one by one until the user entered pattern matches the pattern required to select this entry;
 4. Note that a match is **guaranteed**—no need to keep track of how many items there are
 5. Invoke the function that will act upon the user's wishes.
 6. When the function returns, redraw the menu screen (in case the function has modified the screen) and start over again.
 7. Note that if the user requested to quit the menu entirely, the function of the “quit selection” would call `exit()` and hence never return—so even this event does not have to be considered specially by the menu handler.

Bibliography

[KandR78] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. First Edition.