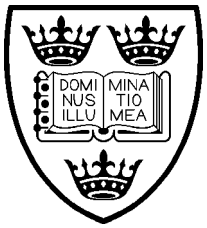
Oxford University Computing Services



Programming in C

Languages l9.2/2

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

Placement Classes

For All Companies placement Material

@placementclasses

CAT

For CAT Exam Preparation Material

@cat_classes

GATE

For GATE Exam Preparation Material

@gate_classes

EBOOK

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C Language

C

@c_examples

C++

@cpp_examples

Java

Java

@java_examples0

Python

@python_examples

Typographical Conventions

Listed below are the typographical conventions used in this guide.

Names of keys on the keyboard are enclosed in angle brackets; for example <Enter> represents the Enter (or Return) key.

Two key names enclosed in angle brackets and separated by a slash (for example, <Ctrl/Z>) indicate that the first key should be held down while the second is pressed; both keys can then be released together.

Where two or more keys are each enclosed in separate angle brackets (for example, <Home><Home><Up arrow>) each key in the sequence should be pressed in turn.

Characters typed in by the user are in lower-case non-bold characters in typewriter font.

Other options and button names are shown in lower-case non-bold characters in typewriter font.

Pull-down menu options are indicated by the name of the option enclosed in square brackets, for example `[File/Print]`. To select the option `[Print]` from the `[File]` menu: click with the mouse button on the `[File]` menu name; move the cursor to `[Print]`; when `[Print]` is highlighted, click the mouse button again.

Where menu items or large sections of output are included, they are shown as they would be displayed on the screen.

Sections marked with a ‡ are more advanced and can be omitted on a first reading.

Contents

References

[1] The C Programming Language, Brian Kernighan and Dennis Ritchie, Second Edition, Prentice Hall, 1988
[2] Writing Solid Code, Steve Maguire, Microsoft Press, 1993
[3] C Traps and Pitfalls, Andrew Koenig, Addison-Wesley, 1989
[4] The Peter Norton Programmer's Guide to the IBM PC, Peter Norton, Microsoft Press, 1985
[5] Rationale for American National Standard for Information Systems – Programming Language – C (see Bibliography for more details).
[6] Programming with ANSI C, Brian Holmes, DP Publications, 1995
[7] Software Developers Kit, Microsoft Press
[8] Mastering C, Anthony Rudd, Wiley-Qed,1994
OUCS Userguides, Oxford University Computing Services.
[9] c9.1/1 Further use of Unix
[10] l9.5/1 Programming in C-Exercise solutions

Original Author: Stephen Gough
Revised by: Brevan Miles

Revision History:

C.T.C	April 1990	Original publication
l9.2/1	April 1994	Second edition
l9.2/2	September 1996	Third edition

1 Introduction

1.1 About this Userguide

This userguide was written to complement the OUCS course *Programming in C*, but also to serve as a reference to the essential aspects of C. It aims to introduce the C language to those already familiar with programming in another language (as covered on *Introduction to programming in Pascal* or *Introduction to programming in Visual Basic* courses). It will show how fundamental programming structures are implemented using the syntax of C.

This guide and the OUCS course will teach ANSI C (see [5]), but where it differs significantly from Kernighan and Ritchie (K&R) C, the K&R method may have been included. It is useful to see the K&R variants, as it is possible that you will encounter K&R code.

Exercises are included in Chapter 18, along with a list of which exercises follow each chapter. A set of solutions to these examples can be found in the OUCS user guide l9.5 Programming in C - Exercise Solutions [10] and are available on the World Wide Web from the URL:
`http:/info.ox.ac.uk/oucs/courseware/c/`

1.2 Why use C?

C (and its object oriented version, C++) is one of the most widely used third generation programming languages. Its power and flexibility ensure it is still the leading choice for almost all areas of application, especially in the software development environment.

Many applications are written in C or C++, including the compilers for other programming languages. It is the language many operating systems are written in including Unix, DOS and Windows. It continues to adapt to new uses, the latest being Java, which is used for programming Internet applications.

C has many strengths, it is flexible and portable, it can produce fast, compact code, it provides the programmer with objects to create and manipulate complex structures (e.g classes in C++) and low level routines to control hardware (e.g input and output ports and operating system interrupts). It is also one of the few languages to have an international standard, ANSI C [5]. The background and advantages of C are covered in more detail in Chapter 15 (see page 65).

2 An Example C Program

```
/* This program prints a one-line message */

#include <stdio.h>

int main()
   {
   printf("Hello World\n");

   return 0;
   }
```

`/* This program ... */`	The symbols `/*` and `*/` delimit a comment. Comments are ignored by the compiler[1], and are used to provide useful information for *humans* that will read the program.
`main()`	C programs consist of one or more functions. One and *only* one of these functions must be called `main`. The brackets following the word `main` indicate that it is a function and not a variable.
`{ }`	braces surround the body of the function, which consists of one or more instructions (*statements)*.
`printf()`	is a library function that is used to print on the standard output stream (usually the screen).
`"Hello World\n"`	is a string constant.
`\n`	is the newline character.
`;`	a semicolon terminates a statement.
`return 0;`	return the value zero to the operating system.

C is case sensitive, so the names of the functions (`main` and `printf`) must be typed in lower case as above.

With a few exceptions, any amount of white space (spaces, tabs and newlines) can be used to make your programs more readable. There are many conventions for program layout, just choose one that suits you, or alternatively get a program to format your code for you (such as the `indent` program).

[1] Actually they are converted into a space in ANSI C; they are deleted completely in K&R C.

3 Variables and Expressions

3.1 Variable Declaration

Variables should be declared either outside a function or at the start of a block of code, after the opening `{` and before any other statements. See section 7.1 for further details. They must be declared before use and the type must normally be specified (exceptions are external and static declarations where the type `int` is implicit, see section 7.2).

```
int       miles, yards;      /* global variables */
main()
{
float     kilometres;     /* local variables */
```

3.2 Variable Types

There are a number of 'built-in' data types in C. These are listed below. Where a shorter version of the type name exists, this is given in brackets; essentially the base type `int` is implicit whenever `short`, `long`, or `unsigned` are used.

`short int`	(`short`)
`unsigned short int`	(`unsigned short`)
`char`	
`unsigned char`	
`signed char`	
`int`	
`unsigned int`	(`unsigned`)
`long int`	(`long`)
`unsigned long int`	(`unsigned long`)
`float`	
`double`	
`long double`	

The range of values that can be stored in variables of these types will depend on the compiler and computer that you are using, but on an IBM PCs and the Borland Turbo C compiler the ranges are:

`short int`	-128 → 127 (1 byte)
`unsigned short int`	0 → 255 (1 byte)
`char`	0 → 255 *or* -128 → +127 [2] (1 byte)
`unsigned char`	0 → 255 (1 byte)
`signed char`	-128 → 127 (1 byte)
`int`	-32,768 → +32,767 (2 bytes)
`unsigned int`	0 → +65,535 (2 bytes)
`long int`	-2,147,483,648 → +2,147,483,647 (4 bytes)

[2] Depends on compiler

`unsigned long int`	0 → 4,294,967,295 (4 bytes)
`float`	single precision floating point (4 bytes)
`double`	double precision floating point (8 bytes)
`long double`	extended precision floating point (10 bytes)

The ANSI C standard states only that the number of bytes used to store a `long int` is equal to or greater than the number used to store an `int`, and that the size of an `int` is at least as big as the size of a `short int`. The C operator, `sizeof`, reports the number of bytes used to store a variable or a type [3], and so the above can be rewritten as:

```
sizeof(long int) ≥ sizeof(int) ≥ sizeof(short int)
```

It may be the case, for your compiler and system, that the sizes for all three are the same.

The same holds for the three floating point types:

```
sizeof(long double) ≥ sizeof(double) ≥ sizeof(float)
```

If you have an ANSI C compiler, the actual ranges for your system can be found in the header files `limits.h` for integer types and `float.h` for floating point types.

ANSI C has the qualifier `signed` which can be applied to any of the integer types (including `char`). `signed` is the default for all the `int` types, but is dependent on the compiler for `char`.

3.3 Variable Names

Variable and function names (or identifiers) consist of a letter followed by any combination of letters and digits. Underscore (`_`) is treated as a letter and it is possible have identifiers that begin with an underscore. However, identifiers beginning with an underscore followed by either another underscore or an upper case letter are reserved, and identifiers beginning with an underscore followed by a lower case letter are reserved for file scope identifiers. Therefore, use of underscore as the first letter of an identifier is not advisable.

Upper and lower case letters are distinct, so `fred` and `Fred` are different identifiers. ANSI C states that the first 31 characters of an identifier are significant except for objects with *external linkage*. External variables are those declared outside of any function, all functions are external; *all* external objects (variables and functions) have *external linkage* unless they are declared to be `static` (see 7.2). As external

[3] `sizeof` can be applied to types and objects; when used with an object, brackets are not needed.

`sizeof(char)` is always 1.

names are used by the linker, which is not part of C, the limits on the number of significant characters are *much* worse, only six characters and no distinction between upper and lower case. Some linkers will be better than this, but it cannot be relied upon. For this reason, the following is dangerous, as the names of the variables do not differ in the first six characters.

```
int integer1;
int integer2;

int main()
   {
   }
```

3.4 Assignment

Assignment is the process of storing a value in a variable, for example:

```
miles = 26;
```

The variable `miles` is known as the *Left Hand Side modifiable value*, i.e. it must be modifiable to be able to store a new value.

3.4.1 Assignment Operators

C has many assignment operators. Pascal and Fortran have *one*.

`=`	assign
`+=`	assign with add
`-=`	assign with subtract
`*=`	assign with multiply
`/=`	assign with divide
`%=`	assign with remainder
`>>=`	assign with right shift
`<<=`	assign with left shift
`&=`	assign with bitwise *AND* (see 3.8)
`^=`	assign with bitwise *XOR*
`|=`	assign with bitwise *OR*

All but the first assignment operator are shorthand methods of modifying a variable or object. For example, the following two statements are equivalent:

```
a = a + 17;
a += 17;
```

The *assign with* operators come into their own where the specification of the object to be modified is long and complicated, and the chance for error in specifying it twice is large. As an example, consider the two statements below, both of which add one to a particular array element:

```
data[abs(nums[x%val])] = data[abs(nums[x%val])] + 1;
data[abs(nums[x%val])] += 1;
```

An expression that refers to a modifiable object *must* appear on the left hand side of each assignment operator. This is often simply the name of a variable, but as we will see later includes structure and union members, dereferenced pointers and elements of arrays. Such expressions are called *modifiable lvalues*.

Assignment operators also produce expressions, thus `a = b` is an expression that has the same value as the value stored in `a` (which is not necessarily the value of `b`!) and has the same type as the type of `a`. This permits statements like:

```
a = b = c = d = e = 0;
```

An example of a situation where the type and value of the assignment expression is *not* the same as the value and type of the right hand operand, consider the following:

```
int main()
   {
   double   x = 1.23, y;
   int      i;

   y = i = x;
   printf("%f\n", y);
   return 0;
   }
```

Here the assignment expression `i = x` has the `int` value `1`, the `double` variable `y` has the value `1.0` after the assignment, *not* `1.23`.

3.5 Arithmetic Operators

`*`	multiplication
`/`	division
`%`	remainder after division (modulo arithmetic)
`+`	addition
`-`	subtraction and unary minus

The `/` operator is used for two different operations: *integer* and *floating point* division. If both operands of the divide operator are of *integral* (`char`, `int` and its derivatives) type then *integer* division is performed. If either operand is `float`, `double` or `long double` then real division is undertaken.

```
int main()
   {
   float a;

   a = 1 / 3;
```

```
    printf("%f\n", a);
    return 0;
    }
```

would print `0.000000` as integer division was performed even though `a` is of type `float`.

3.6 Increment and Decrement Operators

Increment and decrement operators give a shorthand method of adding/subtracting 1 from an object.

`++`	increment
`--`	decrement

These operators can be prefix or postfix. An example of the different behaviour of the prefix and postfix forms is given below, but essentially with the prefix form the variable is changed *before* the value of the expression in which it appears is evaluated, and with the postfix form the variable is modified *afterwards*.

```
b = 3;
a = b++ + 6;          /* a = 9,  b = 4   */

b = 3;
a = ++b + 6;          /* a = 10, b = 4   */
```

3.7 Cast Operators

Cast operators allow the conversion of a value of one type to another.

`(float) sum;`	converts type to `float`
`(int) fred;`	converts type to `int`

3.8 Bitwise Operators ‡

Bitwise operators allow manipulation of the actual bits held in each byte of a variable. Each byte consists of a sequence of 8 bits, each of which can store the value `0` or 1

Operator	**Operation**
~	one's complement
&	bitwise AND
^	bitwise XOR
|	bitwise OR
<<	left shift (binary multiply by 2)
>>	right shift (binary divide by 2)

Examples

Operator	Operand	Result
~	0010111	1101000
<<	0010111	0101110
>>	0010111	0001011

AND	XOR	OR
0 & 0 = 0	0 ^ 0 = 0	0 | 0 = 0
1 & 0 = 0	1 ^ 0 = 1	1 | 0 = 1
0 & 1 = 0	0 ^ 1 = 1	0 | 1 = 1
1 & 1 = 1	1 ^ 1 = 0	1 | 1 = 1

For example, to obtain the two separate bytes of a two byte `int` (`int` may be more than two bytes on your system, and not all computers have eight bit bytes, so this example is not universal):

```
hiByte = (i >> 8) & 0xFF;
loByte = i & 0xFF;
```

If `i = 1011010100110111` then
`i >>8 = 0000000010110101` (shift `I` right by 8 bits).
`(i>>8) & OxFF` ANDs `0000000010110101` with `0000000011111111` giving `10110101` (bits 9-16 of `i`)
`i & 000000011111111 = 00110111` (bits 1-8 of i).
`OxFF` is the hexadecimal representation of `11111111` .

See [6] for more on bitwise operators.

3.9 Promotions and Conversions ‡

3.9.1 Integral Promotions

Any value of type `char` or `short int` (`signed` or `unsigned`) is converted to a value of type `int` (if it can represent all the values of the original type) when it appears in an expression. If `int` cannot store all the values of the original type then `unsigned int` is used.

For example, assuming the following declarations:

```
char  a, b;
```

then

`a + b`	has type `int`
`printf("%c",a);`	the type of the second argument is `int`

```
char  a, b;

a=32;
b=76;
printf("%c",a+b);     /* prints letter l (l=108) */
```

However, if a function prototype (see section 6.6) exists which states that the parameter is `char` then the promotion to `int` will *not* take place, e.g.

```
void fred(char c);

int main()
   {
   char  a;

   fred(a);
   return 0;
   }
```

The argument to `fred` will be of type `char`.

3.9.2 Arithmetic Conversions

Sometimes values within expressions are converted to another type, this is done to preserve information. For example, it is permitted to add a `double` to an `int`, the `int` value will be converted to a `double` before the addition takes place rather than vice versa. For every binary operator in an expression the following rules are used:

- if either operand is `long double` convert the other operand to `long double`
- otherwise, if either operand is `double` convert the other operand to `double`
- otherwise, if either operand is `float` convert the other operand to `float`
- otherwise perform integral promotions on both operands, and then:
  - if either operand is `unsigned long int` convert the other to `unsigned long int`
  - otherwise, if one operand is `long int` and the other is `unsigned int` then:
    - if a `long int` can represent all values of `unsigned int` then convert `unsigned int` operand to `long int`
    - if a `long int` cannot represent all the values of `unsigned int`, then convert both operands to `unsigned long int`
  - otherwise, if either operand is `long int` then convert the other to `long int`
  - otherwise, if either operand is `unsigned int` then convert the other to `unsigned int`
  - otherwise, both operands are of type `int`

3.9.3 Problems

Whilst integral promotions and arithmetic conversions usually act to preserve information it is possible to get caught out. Consider the following on a system where the type of `char` is actually `unsigned char`.

```
#include <stdio.h>

int main()
   {
   char  c = EOF;
   if (c == EOF)
      ...

   return 0;
   }
```

On most systems `EOF` is a preprocessor symbolic constant with the `int` value `-1`. On systems where negative numbers are stored using the two's complement system (this includes the IBM PCs used in class), `-1` is stored by setting every bit in the variable to `1`. When this is copied into `c` above, we end up with a byte of `11111111`. When the comparison is made, one operand is `unsigned char`, the other is `int`. According to the integral promotions rule above, the `unsigned char` value is converted to `int`; but unfortunately as the original type was `unsigned char` the byte of `1s` gets converted to the `int` value `255`. Thus the test becomes `if (255 == -1)` which, not surprisingly, fails. Had the type of `char` been `signed char` then the problem would not have occurred, as the value `-1` would be preserved in the conversion to `int`. In fact, most programmers use `int` to store characters anyway (except in strings).

3.10 Parsing Rules ‡

The C compiler *parses* a C program by splitting up the program into its constituent words and operators. Each separate entity is called a *token*. The C compiler uses a simple rule when deciding where to split the C source. The compiler bites off as large a chunk of your program as possible which will form a legal C token. This mechanism is called *greedy lexical analysis*. Possible ambiguities when using the increment operators can be eliminated if the *greedy* rule is followed.

```
   a = b+++c;      /* a = (b++) + c */
   a = b+ ++c;     /* a = b + (++c) */
```

What about the following?

```
   a = b+++++c;    /* a = (b++) + (++c) */
```

There can be no ambiguity about this statement, the meaning given in the comment is the only sensible one, but the greedy lexical analysis rule means that a C compiler *cannot* arrive at the correct interpretation, instead it thinks the programmer meant

```
a = (b++)++ + c;
```

which is illegal, as the increment operator can only be applied to modifiable *lvalues* of integral type.

3.11 Symbolic Constants and The Preprocessor

```
#define  LIMIT 100
#define  PI    3.14159
#define  NAME  "Steve"
```

The preprocessor allows the creation of symbolic constants (and more — see section 11.6)

`#define LIMIT`	by convention the symbol is given an upper case identifier.
`100`	an `int` constant
`3.14159`	a `double` constant
`"Steve"`	a string constant (again, actually `char *`)

The preprocessor will replace all *tokens* which have been `#define`d by their definition, e.g.

```
printf("%f\n", 2 * PI * radius);
```

becomes

```
printf("%f\n", 2 * 3.14159 * radius);
```

but

```
printf("%s\n", NAMES);
```

does *not* become

```
printf("%s\n", "Steve"S);
```

4 Input and Output

4.1 Formatted Output — *printf*

```
printf("%f km per hour\n", kilometres);
```

In the program statement above:

`"%f km per hour\n"`	is the control string
`kilometres`	is the variable to be printed

`%f` is a conversion specifier indicating that the type of the corresponding variable to be printed is `double`

The onus is on the programmer to ensure both that the number of conversion specifications and the number of variables following the control string are the same (except where `*` is used to specify a field width or precision), and that the conversion character is correct for the type of the parameter.

4.2 Conversion Specifiers

The format of the conversion specifier in a `printf` format string is

1. Mandatory `%` character

2. Optional flags in any order

`-`	print at left of output field
`+`	print number with a sign (even if positive)
space	print a space if the first character to be printed is not a sign character
`0`	pad with leading zeros
#	alternate output format, see Kernighan & Ritchie [1] or manual

3. Optional field width

   The output will take up at least this width on the screen. The output will be padded with spaces (at left if the flag `-` is not present, otherwise at right) if the flag `0` is not present, otherwise padded with zeros. If the output requires more characters than the specified field width it will *not* be truncated. The field width may be specified as `*` in which case the next argument after the format string is used to determine the field width.

4. Optional precision

   Must be preceded by a period. The precision is the number of decimal places of a floating point value to print, or the number of characters of a string. The precision may be specified as `*` in which case the next argument after the format string is used to determine precision.

5. Optional length modifier

h	argument is to be printed as a `short` or `unsigned short` (the argument will have been promoted to `int` or `unsigned int` before being passed to `printf`, see section 3.9.1, the `h` modifier converts it back!)
l	argument is `long` or `unsigned long`
L	argument is `long double`

6. Mandatory conversion character, see 4.2.1.

4.2.1 *printf* Control String Conversion Characters

A table of the control characters used in `printf` statements is given below:

Character	Form of output	Expected argument type
`c`	character	`int`
`d` or `i`	decimal integer	`int`
`x`	hexadecimal integer	`int`
`o`	octal integer	`int`
`u`	unsigned integer	`int`
`e`	scientific notation floating point	`double`
`f`	“normal” notation floating point	`double`
`g`	e or f format, whichever is shorter	`double`
`s`	string	pointer to `char`
`p`	address format (depends on system)	pointer

Arguments of type `char` and `short int` get promoted to `int` and arguments of type `float` get promoted to `double` when passed as parameters to *any* function, therefore use `int` conversion characters for variables of type `char` and `short int`, and `double` conversion characters for a `float` variable or expression.

4.2.2 Examples

Statement	Output	Comment

`char a;`
`a='a';`
`printf("what %c day”,a);`
	`what a day`	variable printed at position of the conversion specifier

`printf("Pi=%f:",PI);`	`Pi=3.142857:`	print `Pi`
`printf("%-10f",PI);`	`Pi=3.142857 :`	print `Pi` in field width of 10, left aligned.
`printf("%10f",PI);`	`Pi= 3.142857:`	print `Pi` in field width of 10, right aligned.
`printf(“Pi=%e:”,Pi);`	`Pi=3.142857e +00`	print `Pi` in scientific notation
`printf("%06.2f",PI);`	`Pi=003.14:`	print `Pi` in field width of `6` with `2` decimal places and padded with leading zeros

4.3 Literal Constants

We need some method of writing literal values in our C programs. The most obvious is the ability to write integer values in decimal form (to base 10); characters, digits and punctuation; floating point numbers with decimal places and strings. C also allows the writing of hexadecimal and octal constants.

Constant	Type
`'Y'`	`int`
`77`	`int`
`77L`	`long`
`77U`	`unsigned`
`77UL`	`unsigned long`
`0.003`	`double`
`0.003F`	`float`
`0.003L`	`long double`
`1.0`	`double`
`0.5e-2`	`double`
`"hello"`	string (actually `const char []` , see 8.9)
`0xA32C`	`int` written in hexadecimal form (41,772)
`017`	`int` written in octal form (15)
`17`	`int` written in decimal form (17)

The suffixes used to indicate `unsigned`, `long`, `unsigned long`, `float` and `long double` can be written in upper or lower case, but watch out for the fact that a lower case `l` (ell) often looks like the digit `1`, and so `L` is preferred.

`double` constants will be shortened to `float` when necessary (for example when assigning to a `float` variable).

Note that the type of a character constant is `int`, not `char`. The value will be the value in the machine's native character set (usually ASCII) corresponding to the character specified.

Also note the difference between `2` and `'2'`. The first is an `int` constant with the value 2, the second is the `int` value of the character `2` in the character set of the computer used (on ASCII machines it is the `int` value 50).

4.3.1 Character Escape Sequences

There are several character *escape* sequences which can be used in place of a character constant or within a string. They are:

Escape sequence	Meaning
`\a`	alert (bell)
`\b`	backspace
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\\`	backslash
`\?`	question mark
`\'`	quote
`\"`	double quote
`\`*`ooo`*	character specified as an octal number
`\x`*`hh`*	character specified in hexadecimal

4.4 Formatted Input — *scanf*

`scanf` is used to interpret characters input to the computer and to store the interpretation in the specified variable(s).

`scanf("%d", &x);`	read a decimal integer from the keyboard and store the value in the memory address of the variable `x`

The `&` character is vital with arguments that are not pointers. (See section 8).

4.4.1 *scanf* Control String Conversion Characters

`scanf` uses the same conversion characters as `printf`.

The arguments to `scanf` *must* be pointers (addresses), hence the need for the `&` character above.

Character	Form of output	Expected argument type
c	character	pointer to `char`
d	decimal integer	pointer to `int`
x	hexadecimal integer	pointer to `int`
o	octal integer	pointer to `int`
u	unsigned integer	pointer to `int`
i	integer	pointer to `int`
e	floating point number	pointer to `float`
f	floating point number	pointer to `float`
g	floating point number	pointer to `float`
s	string	pointer to `char`
p	address format, depends on system	pointer to `void`

The conversion characters `e`, `f` and `g` have exactly the same effect, they read a floating point number which can be written with or without an exponent.

When reading integers using the `i` conversion character, the data entered may be preceded by `0` or `0x` to indicate that the data is in octal (base 8) or hexadecimal (base 16).

The `d`, `x`, `o`, `u` or `i` conversion characters should be preceded by `l` if the argument is a pointer to a `long` rather than an a pointer to an `int`, or by `h` of the argument is a pointer to short `int`. The `e`, `f`, or `g` conversion characters should be preceded by `l` if the argument is pointer to `double` rather than pointer to `float`, or `L` for pointer to `long double`.

Maximum field widths can be specified between the `%` and the conversion character. For example, the following call to `scanf` will read no more than 50 characters into the string variable `str`.

```
scanf("%50s",str);
```

A * between the % and the conversion character indicates that the field should be skipped.

Spaces and tabs in the format string are ignored. scanf also skips leading white space (spaces, tabs, newlines, carriage returns, vertical tabs and formfeeds) in the input (except when reading single characters). However a subsequent space terminates the string.

```
char a[50];
scanf("%s",&a);       /* Input “Where is it” */
puts(a);              /* Outputs “Where” */
```

To read the other strings add more conversion characters or use gets() (see 0)

```
scanf(“%s%s%s”,&a,&b,&c);
```

Other characters that appear in the format string must be matched exactly in the input. For example, the following scanf statement will pick apart a date into separate day, month and year values.

```
scanf("%d/%d/%d",&day,&month,&year);
```

It is also possible to use a *scan set* when reading input, the scan set lists the only characters that should be read (or should not be read) from the input stream for this conversion, for example:

`scanf("%[a-z]c",&c);`	only read lower case alphabetic characters
`scanf("%[^0-9]c",&d);`	do not read digits

4.5 Character I/O — *getchar* & *putchar*

getchar and putchar are used for the input and output of single characters respectively.

`getchar()`	returns an int which is *either* EOF (indicating end-of-file, see 4.6) or the next character in the standard input stream
`putchar(c)`	puts the character c on the standard output stream.

```
int main()
   {
   int    c;

   c = getchar();     /* read a character and assign to c */
   putchar(c);        /* print c on the screen */

   return 0;
   }
```

4.6 End-of-File

It is important that we are able to detect when no more data is available on the standard input stream. If standard input has been redirected from the contents of a file on disk, then the "end-of-file" condition is when we have read all the data stored in that file. It is also possible for a user to indicate end-of-file when standard input is left connected to the keyboard; though the exact mechanism varies between different operating systems. On MS-DOS, the user should type `<Ctrl/Z>`, on Unix `<Ctrl/D>`. To aid portability between different operating systems, C provides an end-of-file value, which is written `EOF`. This value will be returned by the input routines `scanf` and `getchar` on end-of-file. The value of `EOF` is defined in the header file `stdio.h`, and it is necessary to include this file before referring to the constant value `EOF`. This is done by adding the *preprocessor* directive

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

before any statement where the identifier `EOF` is used. Normally, however, such include commands are given at the beginning of the program.

5 Flow of Control

5.1 Relational and Logical Operators

The following relational operators produce a true or false value. True and false, when generated by such an expression, are represented by the `int` values `1` and `0` respectively. Thus the expression

```
1 < 10
```

has the `int` value `1`. Note, however, that C treats any non-zero value as being true!

Operator	Meaning	Precedence
>	greater than	2
>=	greater than OR equal	2
<	less than	2
<=	less than OR equal	2
==	equal	1
!=	not equal	1

Operators with higher precedence number get evaluated first (a complete list of C operators and their relative precedence is given in section 19).

The following logical operators are used to combine logical values.

`&&`	AND
`||`	OR
`!`	NOT

5.2 Conditional Branching — *if*

In all of the following examples `statement` *can be replaced by a compound statement (or block) which consists of several statements enclosed within braces.*

```
if (expression)   /* if expression is true    */
   statement1;    /*    do statement1         */
else              /* otherwise                */
   statement2;    /*    do statement2         */
```

The `else` part of the statement is optional. Note (especially Pascal programmers) that the semicolon terminating the first statement is required, even if the `else` part is present.

The whole construct above can be treated as a single statement in places where a statement is expected, in particular an `if-else` statement can appear nested inside another `if-else` statement. When nesting statements in this way, it should be noted that an `else` always matches with the nearest unmatched `if`. Thus, in the example below, `statement2` will be executed only when `a` < `b` *and* `c` ≥ `d`.

```
if (a < b)
   if (c < d)
      statement1;
   else
      statement2;
```

If it is required to match an `else` with an earlier `if` than the nearest, then braces are required. For example:

```
if (a < b)
   {
   if (c < d)
      statement1;
   }
else
   statement2;
```

`statement2` is now executed when `a` ≥ `b`.

5.3 Conditional Selection — *switch*

```
switch ( expression)
   {
   case value : statement; statement; ...
   case value : statement; statement; ...
   .
   .
   default : statement; statement; ...
   }
```

`switch` is a mechanism for jumping into a series of statements, the exact starting point depending on the value of the expression. In the example below, for example, if the value `3` is entered, then the program will print `three two one something else!`

```
int main()
   {
   int    i;

   printf("Enter an integer: ");
   scanf("%d",&i);

   switch(i)
      {
      case 4:  printf("four ");
      case 3:  printf("three ");
      case 2:  printf("two ");
      case 1:  printf("one ");
      default: printf("something else!");
      }

   return 0;
   }
```

This may not be what was intended. This process of executing statements in subsequent `case` clauses is called *fall through*. To prevent fall through, `break` statements can be used, which cause an immediate exit from the `switch` statement. In the example above, replacing

```
      case 4:  printf("four ");
```

with

```
      case 4:  printf("four "); break;
```

and adding `break` statements to the statements for the other labels, will result in a program that prints only one string depending on the value of the integer input by the user.

The values listed in the `case` part of the `switch` statement must be constant integer values; integer expressions can be used as long as the value can be determined at compile time.

The `default` label is optional, but if it is present acts as a *catch all* clause.

The labels can occur in any order; there is no need to have the `default` label last, for example (but it usually reads better if it is!).

5.4 Iteration — *while, for*

```
while (expression)      /* while expression is true do*/
   statement;           /*    statement
   */

do                      /* do                          */
   statement;           /*    statement                */
while (expression);     /* while expression is true
   */

for (expr1; expr2; expr3)
   statement;

expr1;                  /* equivalent (almost) to above    */
while (expr2)           /* for loop                        */
   {
   statement;
   expr3;
   }
```

The difference between the `while` and the `do-while` loops is the location of the test. With a `while` loop the test is made before the statement that forms the body of the loop is executed; it is possible that the statement is *never* executed. With the `do-while` loop, the statement will always be executed at least once; the value of the expression then determines if the statement is executed again.

`for` loops behave (more or less) like the equivalent `while` loop shown in the generic example above. The first expression is executed as a statement first, the second expression is then tested to see if the body of the loop should be executed. The third expression is executed at the end of every iteration of the loop.

An example `for` loop, to execute a statement 10 times[4], is given below:

```
for (i = 0; i < 10; i++)
   printf("%d\n",i);
```

[4] An alternative way of executing a statement *n* times is:

```
i = n;
while (i--)
   statement;
```

If you use this method, make sure that *n* is greater than zero, or make the test `i-- > 0`.

Any of the three expressions in the `for` loop can be omitted. Leaving out the first or the third expressions means that no additional action is performed either before the loop starts or after each iteration. Omitting the second expression results in a loop that will always execute, the value of the controlling expression is assumed to be true. If any expression is omitted, the separating semi-colons must still be included. The following `for` loop does nothing before it starts, performs no additional action after each iteration and will continue forever!

```
for (;;)              /* do forever */
   printf("hello\n");
```

The following two statements can be used within any of the three loops we have seen to add additional control over the execution of the loop. Normally, they are used to abort a process if something has gone wrong.

`break`	exits immediately innermost enclosing loop
`continue`	go immediately to next iteration of loop

When `continue` is used in a `for` loop, the third expression in the control statement is guaranteed to be executed. This behaviour is different from the "equivalent" `while` loop.

5.5 Local Jumps — *goto* ‡

It is possible to jump to any statement within the *same* function using `goto`. A label is used to mark the destination of the jump. `goto` is rarely, if ever, needed, as `if`, `switch`, `while` and `for` should provide all the branching and iteration structures needed. However, there are times when a `goto` simplifies the code greatly. A frequently cited example is when something goes disastrously wrong deep within nested loops:

```
void fred(void)
   {
   while ( ... )
      for( ... )
         if (disaster) goto error;

  error:
   tidy up the mess
   }
```

But in the example above, remember that the code could be rewritten as:

```
void fred(void)
   {
   while ( ... )
      for( ... )
         if (disaster)
            {
```

```
            tidy up the mess
            return;  /* and get out of here! */
            }
   }
```

5.6 Short Circuit Behaviour ‡

Whenever expressions are connected together using the logical operators `&&` (AND) or `||` (OR), only as many expressions as are needed to determine the overall logical value will be evaluated. This is known as *short-circuit* behaviour. For example, if two expressions are connected with `&&` and the first expression is false, then it is guaranteed that the second expression will not be evaluated. The same is true of expressions connected with `||` where the first expression is true. For example:

```
if (b != 0 && a / b > 1)
   statement;
```

will prevent the evaluation of `a / b` if `b` is zero.

If the programmer accidentally uses `&` instead of `&&` then the short-circuit behaviour is lost, which, in the example above may lead to a run-time division by zero error.

5.7 Problems

5.7.1 && c.f. &

Unfortunately it is very easy to forget that the logical AND operator is represented by two ampersands, and type only one by mistake (which is the bitwise AND operator). The problem is made worse by the fact that the net result is often the same! The root of the problem is that the value of true (when generated by a relational or logical operator) is `1` and false is `0`, but C treats any non-zero value as being true. So, as an example, consider the two expressions:

(a) `(a > b) && (c < d)`

(b) `(a > b) & (c < d)`

The values of the two expressions are:

`a > b`	`c < d`	(a)	(b)
false(0)	false(0)	0	0
false(0)	true(1)	0	0
true(1)	false(0)	0	0
true(1)	true(1)	1	1

In contrast, consider:

```
if ( fred() && jim() )
```

where the functions return `int` values. If `fred` returns the value `2` and `jim` the value `1` (both of which are treated as being true), then the test above is true. If `&` is used by mistake, then the test will be the result of a bitwise AND of `2` (10) and `1` (01), which is `0` (00) or false.

5.7.2 Assignment c.f. Comparison

`a = 1;`	1 is assigned to the variable a
`a == 1;`	variable `a` is compared with the constant `1`

It is very easy to type `=` by mistake instead of `==`. However, the problem is compounded by the fact that most of the time the compiler will not pick up your mistake. Why?

(a)

```
while (a == 6)        /* while a is equal to 6 do the */
   statement;         /*    statement */
```

(b)

```
while (a = 6)
   statement
```

Code (a) is probably the intended statement. What happens if you accidentally type (b) instead. The code given in (b) is perfectly legal C. What does it do?

- 6 is assigned to `a`.
- the expression `a = 6` is tested — it has the value `6` which is non-zero and therefore true.
- the loop will be executed forever as `a = 6` is always true.

To avoid such problems, some programmers write the constant on the left hand side of the `==` operator, then if they accidentally type `=` instead, the compiler will produce an error as a constant is not an lvalue. This will only work for comparison with a constant or expression.

5.8 Declaring Array Variables

The following declaration states that `x` is an array of three integer values.

```
int      x[3];
```

Each individual part (or *element*) of `x` is accessed by adding an index value in square brackets after the array name, e.g.

```
x[0] = 74;
printf("%d\n", x[2]);
```

Note that the index values range from `0` to one less than the number of elements.

Arrays can have any number of dimensions; to declare a 2-D array of `double`:

```
double   matrix[10][10];
```

5.9 Initialising Array Variables

```
int      ndigit[10]  = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
char  greeting1[]  = "hello";
char  greeting2[]  = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

The last two declarations show the two alternative ways of initialising an array of characters. They are exactly equivalent, but one is somewhat easier to type!

If the number of elements of the array is not specified, then the compiler will determine this from the initialiser. The declarations of `greeting1` and `greeting2` above both declare an array of 6 characters.

The values given in a brace delimited initialiser (first and last examples above) *must* be constant expressions (one whose value can be determined at the time of compilation). If the initialiser is a single expression, the restriction does not apply.

If the number of expressions in the initialiser is less than the number of elements in the array then the remaining elements will be initialised as though they were *static* objects; that is arrays of numeric values will be initialised with zeros whatever the storage class of the array (external, static or automatic).

When initialising multi-dimensional arrays, extra braces can be used to show the reader how the values are to be used:

```
int   x[3][2]  = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
```

Extra braces are needed if values are to be omitted from the initialiser:

```
int   x[3][2]  = { { 1, 2 }, { 3 },  { 5, 6 } };
```

In this example we have omitted the initialiser for `x[1][1]`; without the extra braces the compiler would initialise `x[1][1]` with `5` and would leave `x[2][1]` (i.e. the last element) uninitialised.

K&R C permits initialisation of static arrays only.

6 Functions

6.1 Building Blocks of Programs

All C programs consist of one or more functions. Functions are the building blocks of a program. All functions are at the same level — there is no nesting. One (and only one) function must be called `main`.

6.2 Return Value

All functions can return a value, including `main`.

Functions can return arithmetic values (`int`, `float` etc.), structures, unions, pointers or `void`. If the return type is specified as being `void`, then no value is returned by the function. Functions cannot return a function or an array.

For functions that will return a value, the `return` statement is used in conjunction with an expression of an appropriate type. This causes immediate termination of the function, with the value of the expression being returned to the caller. `return` can also be used with functions of type `void`, but no return value can be specified.

6.3 Function Parameters

All functions (including `main`) can accept parameters.

The example below shows the old Kernighan & Ritchie C function definition format.

```
double minimum(x, y)
   double x, y;
   {
   if (x < y)
      return x;
   else
      return y;
   }
```

ANSI C will accept this old style definition, but introduces a newer, safer, definition format:

```
double minimum(double x, double y)
   {
   if (x < y)
      return x;
   else
      return y;
   }
```

In both cases the *parameter* list must be specified, and their types must be declared. The parameters to the function (the expressions given in the function call) are passed by *value* only.

If, in a new style definition, the parameter list contains the single word `void`, then the function does not take any parameters.

It is common to call the variables specified in the function definition *parameters* and the expressions given in a function call *arguments*. For example, in the following call of `minimum`, the expressions `a` and `b * 2` are the arguments to the function. The values of the two expressions will be copied into the parameters `x` and `y`. Sometimes the terms *formal argument* and *actual argument* are used instead; the formal argument being the variable given in the function definition, the actual argument being the expression given in the function call.

```
   minimum(a, b*2);
```

6.4 Variable Function Parameters

All function parameters are passed by value. To make a function alter a variable, the address of the variable must be passed, i.e. pass the variable by reference.(See 8)

```
int max(int a, int b, int *c);

int main()
   {
   int x = 4, y = 5, z;
   max(x, y, &z);        /* generate a pointer to z */
   return 0;
   }

int max(int a, int b, int *c)
   {
   if (a > b)
      *c = a;            /* *c modifies the variable    */
   else                  /* whose address was passed    */
      *c = b;            /* to the function             */
   }
```

6.5 Function Definition and Declaration

A function *definition* is where the function name, parameters, code and return type are specified. A function *declaration* is where the name and return type of a function are given. The definition of a function includes a declaration of that same function implicitly.

A function can be declared many times (as long as the declarations declare the function to be of the same type) but can only be defined once. Declarations of functions are sometimes necessary to appease the compiler, which always assumes, if the information is not available, that all functions return `int`.

```
/* declaration of minimum() */
double minimum(double x, double y);
```

```
int main()
   {
   printf("%f\n", minimum(1.23, 4.56));
   return 0;
   }

/* definition of minimum()  */
double minimum(double x, double y)

   double      x, y;
   {
   if (x < y)
       return x;
   else
       return y;
   }
```

The problem could also be solved by placing the function definition before the call to the function.

6.6 Function Prototypes

The ANSI C standard introduces function prototypes. An example is given below. It also allows function definitions to be written in the same form as the new prototypes.

```
double minimum(double, double);
                                /* prototype  of minimum() */

int main()
   {
   printf("%f\n", minimum(1.23, 4.56));
   return 0;
   }

double minimum(double x, double y)
                                /* definition of minimum() */
   {
   if (x < y)
       return x;
    else
       return y;
   }
```

The use of prototypes or the new style function definition allows the compiler to check that the parameters to a function are sensible (not necessarily the same), as well as checking the return type of the function; but only if the definition or prototype appears *before* the function call.

ANSI C draws a distinction between the following two statements. The first is a function *declaration* stating that `fred` takes an, as yet, unspecified number of parameters. The second is a function *prototype* which states that `jim` takes no parameters (see also 7.3).

```
double fred();        /* declaration */
double jim(void);     /* prototype */
```

6.6.1 Standard Header Files

Prototypes of the library functions are given in several standard header files. For example, `stdio.h` contains prototypes for `printf`, `scanf`, `putchar` and `getchar`. Other standard header files are:

`assert.h`	assertions
`ctype.h`	character class tests
`float.h`	system limits for floating point types
`limits.h`	system limits for integral types
`math.h`	mathematical functions
`setjmp.h`	non-local jumps
`signal.h`	signals and error handling
`stdarg.h`	variable length parameter lists
`stdlib.h`	utility functions; number conversions, memory allocation, `exit` and `system`, Quick Sort
`string.h`	string functions
`time.h`	date and time functions

To include these standard header files in your code, use the preprocessor directive `#include` and place angle brackets around the name of the file, e.g.

```
#include <math.h>
```

7 Scope, Blocks and Variables

7.1 Blocks and Scope

C is a block structured language. Blocks are delimited by `{` and `}`. Every block can have its own local variables. Blocks can be defined wherever a C statement could be used. No semi-colon is required after the closing brace of a block.

```
{
int         a = 5;
printf("\n%d", a);
   {
   int      a = 2;
   printf("\n%d", a);
   }
}
```

Reference to a variable will be to the variable of that name in the nearest enclosing block.

7.2 Variable Storage Classes ‡

`auto`	The default class. Automatic variables are local to their block. Their storage space is reclaimed on exit from the block.
`register`	If possible, the variable will be stored in a processor register. May give faster access to the variable. If register storage is not possible, then the variable will be of automatic class. Use of the register class is not recommended, as the compiler should be able to make better judgement about which variables to hold in registers, in fact injudicious use of register variables may slow down the program.
`static`	On exit from block, static variables are not reclaimed. They keep their value. On re-entry to the block the variable will have its old value.
`extern`	Allows access to external variables. An external variable is either a global variable or a variable defined in another source file. External variables are defined outside of any function. (Note: Variables passed to a function and modified by way of a pointer are not external variables)
`static external`	External variables can be accessed by any function in any source file which make up the final program. Static external variables can only be accessed by functions in the same file as the variable declaration.

7.3 Declaration versus Definition

Definition is the place where variable is created (allocated storage).

Declaration is a place where nature (type) of variable is stated, but no storage is allocated.

Variables can be declared many times, but defined only once.

```
int            v;       /* definition of v */

int max(int a, int b)
   {
   extern int   v;       /* declaration of v */
   ....
   }
```

Declaring a variable without defining it is more useful when a variable needs to be accessed in more than one source file when a program is split between multiple source files. For example:

a.c

b.c

```
int v;    /* declaration and
             definition */

a()
   {
   v = 15;
   }
```

```
extern int v;   /* declaration
                   only */

b()
   {
   printf("%d\n",v);
   }
```

In fact, as variables can be declared more than once in the *same* source file, and to ensure that the variable is declared identically in each source file, it is common to place declarations of global variables in a *header* file which is then *included* in each source file.

prog.h

```
extern int   v;
```

a.c

b.c

```
#include "prog.h"

int v;    /* definition */

a()
   {
   v = 15;
   }
```

```
#include "prog.h"



b()
   {
   printf("%d\n",v);
   }
```

7.4 Initialisation of Variables ‡

If variables are not explicitly initialised, then external and static variables are initialised to zero; pointers (see 8) are initialised to `NULL`; `auto` and `register` variables have undefined values.

```
int       x = 1;
char      quote = '\'';
long      day = 60 * 24;
int          len = strlen(s);
```

`external` and `static`	initialisation done once only
`auto` and `register`	initialisation done each time block is entered

`external` and `static` variables cannot be initialised with a value that is not known until run-time; the initialiser must be a constant expression.

8 Arrays, Pointers and Strings

8.1 Pointers are Addresses

A pointer is really the address of something in memory.

Pointers are needed if it is required that a function change the value of an object (though global variables could be used). For example, the `scanf` function takes pointers as arguments (See 6.4).

8.2 Pointers are not Integers

It is possible to treat pointers as though they are integers since they hold actual addresses in memory e.g 7232. `printf`, `scanf`, assignment and some of the arithmetic operators work with pointers, but in many cases such operations should not be used as the results will differ from one machine to another. It is not, for example, guaranteed that a pointer will be the same size as an `int`. In fact, even on an IBM PC the size of a pointer variable depends on the memory model used, and so the assumption that it occupies two bytes is an extremely dangerous one.

8.3 The * and & Operators

`&` gives the address of something in memory, that is it generates a *pointer* to the object

`*` gives what is pointed at by a pointer (called *dereferencing*)

For example, assuming the following declaration:

```
int i = 17;
```

Then the expression `&i` generates a *pointer* to the variable `i`. We can then find out what this pointer is pointing at by applying the `*` operator to the newly generated pointer; thus `*&i` should have the value 17.

8.4 Declaring Pointer Variables

To declare a pointer variable we must state the type of object that the pointer will point at. To declare a pointer to an integer, we write:

```
int *ptr;
```

which says that the type of `*ptr` is `int`, that is the object that we find when following the pointer `ptr` is an integer. Therefore the type of `ptr` is *pointer to* `int`, or `int *`.

When declaring several pointers in a comma separated list, it is necessary to place a `*` before the identifier of *each* pointer:

```
int *ptr1, *ptr2, *ptr3;
```

this declares three pointers to integers, whereas

```
int *ptr1, ptr2, ptr3;
```

declares one pointer variable and two integer variables.

When using initialisers with pointer variables, it is the variable itself that is being initialiased (i.e the address the pointer holds) and not anything at the other end of the pointer:

```
main()
   {
   int i = 17;         /* ordinary int variable */
   int *pi = &i;       /* a pointer to an integer
                          initialis ed to point to i
                          pi now holds address of i */

   *pi = 25;           /* assign to what is pointed at
                          by the pointer,
                          i now contains the value 25*/

   printf("%d",*pi)    /* prints the value of what is in
                          the address pi contains e.g 25*/
   printf("%d2,pi)     /* prints the contents of pi
                          e.g 6582,    (the address of i) */
   return 0;
   }
```

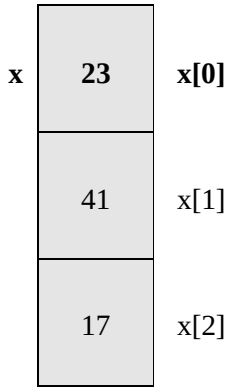
8.5 Pointers and Arrays

An array name behaves in many ways as though it is a pointer to the first element of the array.

However, the name of an array is a constant, it is not possible to change its value, otherwise it would be possible to lose access to the contents of the array. Also, an array name is not an *lvalue*, it cannot appear on the left hand side of an assignment operator.

The following array declaration gives the following structure in memory:

```
int x[3] = { 23, 41, 17 };
```



Expression	Value
`x[0]`	23
`x[1]`	41
`x`	an address, where the data is stored
`*x`	23
`x+1`	an address, where the second element is stored
`*(x+1)`	41
`x+2`	an address, where the third element is stored
`*(x+2)`	17
`&(x[0])`	`x`
`*&(x[0])`	23
`&*(x[0])`	*illegal*, the `&` and `*` do not simply "cancel" [This would be a valid expression if the array was an array of pointers!]

The operation of accessing what a pointer is pointing at via the `*` operator is called *dereferencing* the pointer or *pointer indirection*.

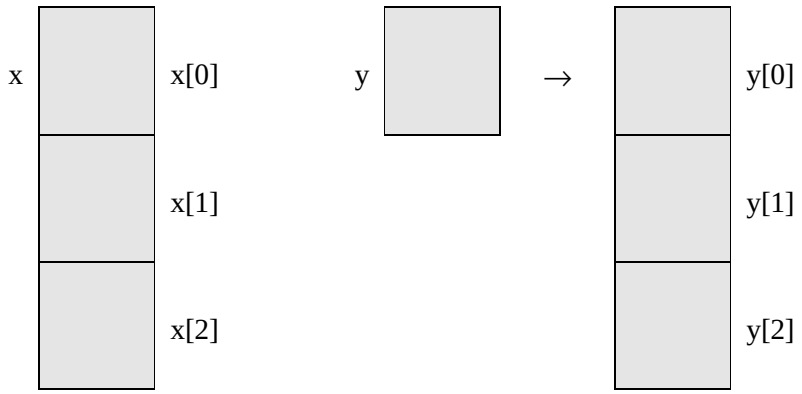
8.6 Dynamically Sized Arrays

Pointers can be used to produce dynamically sized arrays.

Assuming the following declarations:

```
int x[3];
int *y = malloc(3*sizeof(int));
```

This gives the following structures in memory:



The confusion is compounded by the fact that array notation can be used to access the memory pointed to by `y`, thus `y[0]`, `y[1]` and `y[2]` access the memory allocated with the call to `malloc()`.

However, on an IBM PC running MS-DOS and using the *small* memory model `sizeof(x)` is 6 and `sizeof(y)` is 2.

8.7 The *NULL* Pointer and Pointer to *void*

The constant `0` when used in a pointer context (e.g. assigned to a variable of any pointer type or compared with a pointer value) is replaced by a null pointer of the appropriate type. Such a pointer is used to indicate that a pointer does not point anywhere or is invalid. The preprocessor constant `NULL` (defined in the header file `stdio.h`) can be used instead of the constant `0`.

Library functions that return pointers return `NULL` if an error occurs. The value of a pointer should always be checked to ensure that it is not `NULL` before attempting to dereference it. Attempting to dereference a `NULL` pointer will give a run-time error on some systems.
Pointer to `void` (`void *`) is ANSI C's generic pointer type (in K&R C it is `char *`). Functions, such as `malloc()`, which return a generic pointer, return `void *`. Values of type `void *` can be assigned to any other pointer type without a cast, but this is potentially dangerous, as the object the pointer references may not be of the same type as the pointer to which it is being assigned.

```
char *a;
void *b;
b="Hello";
a=b;
printf("%s",a);
```

8.8 Pointer Arithmetic

```
char     *p, s[100];
int      *a, b[100];
double   *f, g[100];

p = s;   /* p points to s[0] */
a = b;   /* a points to  b[0] */
f = g;   /* f points to g[0] */
p++;     /* p points to s[1] */
a++;     /* a points to b[1] */
f++;     /* f points to g[1] */
```

Incrementing a pointer will always step the pointer the appropriate number of bytes so that it skips onto the next object in memory, i.e. incrementing a pointer to a `float` will move the pointer onto the next `float`.

8.9 Strings

A string is a contiguous sequence of `char` values terminated by the `NULL` character (written as `'\0'`). Such character sequences can be stored in an array of `char` or accessed via a pointer. If the latter approach is taken then care is needed ensure that memory needed to store the string itself is allocated if needed.

The following shows the two alternative ways of declaring a string variable capable of storing up to *50* characters (remember that the string needs to be terminated by `'\0'`, space *must* be reserved for this character). The first element of the string `s` is always `s[0]` and `s` is a pointer to the first element in the array.

```
char     s1[50];
char     *s2 = malloc(50);     /* s2 contains the address of
                                  s2[0] */
```

Note, however, that the call to `malloc()` may fail, so some run-time checking of the value of `s2` (to make sure that it is not `NULL`) should be included.

A common problem when using pointers to store strings is to forget to allocate any memory to store the string itself, thus:

```
char     *str;
```

allocates (on MS-DOS computers with the small memory model) two bytes of memory in which to store the pointer to the first character of the string. No other memory is reserved. If an attempt is made to store a string via this pointer, perhaps by reading a string from the standard input stream:

```
scanf("%49s", str);
```

then the program may crash. The reason is that `scanf` will simply store characters in consecutive memory locations starting at the current value of `str`. In this case `str` was never initialised and therefore will contain some random value. `scanf` may well be trying to write the characters read into memory where other variables are stored, where the program itself is stored, or perhaps even where the operating system is stored!

Whenever literal strings are used in a program, only the starting memory address of the stored string is saved, the terminating `'\0'` will mark the end. Thus

```
printf("%s", "hello world");
```

will print `hello world`, whilst

```
printf("%p", "hello world");
```

will print the memory location where the string is stored.

It is useful to investigate the types and meanings of the various pointer operators when applied to a string accessed through a pointer. For example:

```
char  *s = "hello world";
```

declares a pointer to a `char` called `s`, and initialises it to point to the string `"hello world"`. Remember that the type of `*s` is `char`, therefore the type of `s` is `char *` or pointer to `char`.

	meaning	**type**
`&s`	address of the pointer variable `s`	pointer to pointer to `char`, (`char **`)
`s`	value of `s`, where the string `"hello world"` is stored	pointer to `char`, (`char *`)
`*s`	first character of the string (`'h'`)	`char`
`s[0]`	first character of the string (`'h'`)	`char`

8.9.1 Library String Functions

The ANSI standard library includes the following string functions.

```
char  *gets(a)          reads a from standard input
int   puts(a)           prints a to standard output
char  *strcpy(a, b)     copies b to a, returns a
char  *strcat(a, b)     appends b onto a, returns a
int   strlen(a)         returns length of a
char  *strchr(a, c)     returns pointer to first occurrence of c
                        in a
```

`char *strrchr(a, c)`	returns pointer to last occurrence of `c` in `a`
`int strcmp(a, b)`	compares strings, returns `< 0` if `a < b`, `> 0` if `a > b` and `0` if `a == b`

`a` and `b` are strings, `c` is a `char`.

Prototypes for these functions are in the header file `string.h`, include this file for proper checking of arguments to these functions.

It is the programmer's responsibility to ensure that enough memory has been allocated to store the new strings that are formed by the functions `strcpy` and `strcat`.

`strcmp` performs a lexicographic comparison of the two string arguments using the character set of the host computer, thus on an ASCII machine `"Fred"` is less than `"file"` (`'F'` occurs before `'f'` in the ASCII character set) and `"100"` is less than `"2"`. If you wish to perform string checking using some collating sequence other than that given by the base character set, then use `strcoll` instead.

The following program illustrates use of the library string functions and some of the problems with using string variables in C:

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

main()
   {
   char  s1[100] = "hello";    /* initialise an array of
                               characters */
   char  s2[100] = "world";
   char  s3[100];
   char  *s4 = "HELLO";        /* initialise pointer s4
                               to point to the string
                               "HELLO" */
   char  *s5;
   char  *s6;

   strcpy(s3,s1);              /* copy "hello" into s3 */
   strcat(s3," ");             /* append " " */
   strcat(s3,s2);              /* append "world" */
   printf("%s\n",s 3);         /* should print "hello world" */
   strcpy(s5,s4);              /* BAD idea, copies 'h', 'e' ...
                               into memory location starting
                               at the current value of s5,
                               s5 was never initialised and
                               so contains some undefined value;
                               the program may crash */

   s5 = malloc(strlen(s4)+1);
                               /* MUCH better, set s5 to point
                               enough memory to store the
                               string;
                               don't forget that we need to
```

```
                            reserve space for the '\0' too,
                            hence `strlen(s4)+1' */
    strcpy(s5,s4);

    s6 = strdup(s4);        /* equivalent to malloc/strcpy
                            combination above; but NOT ANSI
                            so not always available */

    gets(s5);               /* read in s5 from stdin */

    puts(s5);               /* writes s5 to stdout */

    free(s5);               /* not really necessary as heap
                            will be reset on program
                            termination */
    free(s6);
    }
```

9 Files

9.1 File Pointers

There are many ways to use files in C. Many of them mirror the operating system use of files.

The most straightforward use of files is via a file pointer.

```
FILE  *fp;
```

`fp` is a pointer to a file.

The type `FILE`, is not a basic type, instead it is defined in the header file `stdio.h`, this file *must* be included in your program.

9.2 Opening a File

```
fp = fopen(filename, mode);
```

The filename and mode are both strings.

The mode can be

`"r"`	read
`"w"`	write, overwrite file if it exists
`"a"`	write, but append instead of overwrite
`"r+"`	read & write, do not destroy file if it exists
`"w+"`	read & write, but overwrite file if it exists
`"a+"`	read & write, but append instead of overwrite

`"b"` may be appended to any of the above to force the file to be opened in binary mode rather than text mode

`fp = fopen("data.dat","a");` will open the disk file `data.dat` for writing, and any information written will be appended to the file

The following useful table from the ANSI C Rationale [5] lists the different actions and requirements of the different modes for opening a file:

	r	w	a	r+	w+	a+
file must exist before open	✓			✓		
old file contents discarded on open		✓			✓	
stream can be read	✓			✓	✓	✓
stream can be written		✓	✓	✓	✓	✓
stream can be written only at end			✓			✓

`fopen` returns `NULL` if the file could not be opened in the mode requested. The returned value should be checked before any attempt is made to access the file. The following code shows how the value returned by `fopen` might be checked. When the file cannot be opened a suitable error message is printed and the program halted. In most situations this would be inappropriate, instead the user should be given the chance of re-entering the file name.

```
#include <stdio.h>

int main()
   {
   char  filename[80];
   FILE  *fp;

   printf("File to be opened? ");
   scanf("%79s", filename);

   fp = fopen(filename,"r");
   if (fp == NULL)
      {
      fprintf(stderr, "Unable to open file %s\n", filename);
      return 1;       /* Exit to operating system */
      }

   code that accesses the contents of the file

   return 0;
   }
```

Sequential file access is performed with the following library functions.

`fprintf(fp, formatstring, ...)`	print to a file
`fscanf(fp, formatstring, ...)`	read from a file
`getc(fp)`	get a character from a file
`putc(c, fp)`	put a character in a file
`ungetc(c, fp)`	put a character back onto a file (only one character is guaranteed to be able to be pushed back)
`fopen(filename, mode)`	open a file
`fclose(fp)`	close a file

The standard header file `stdio.h` defines three file pointer constants, `stdin`, `stdout` and `stderr` for the standard input, output and error streams. It is considered good practice to write error messages to the standard error stream (so that error messages do not get redirected to files with the Unix and MS-DOS `>` redirection operator, or disappear down pipes). Use the `fprintf()` function to do this:

```
fprintf(stderr,"ERROR: unable to open file %s\n", filename);
```

The functions `fscanf()` and `getc()` are used for *sequential* access to the file, that is subsequent reads will read the data following the data just read, and eventually you will reach the end of the file. If you want to move forwards and backwards through the file, or jump to a specific offset from the beginning, end or current location use the `fseek()` function (though the file *must* be opened in binary mode). The function `ftell()` reports the current offset from the beginning of the file.

If you wish to mix reading and writing to the same file, each switch from reading to writing (or vice versa) *must* be preceded by a call to `fseek()`, `fsetpos()`, `rewind()` or `fflush()`. If it is required to stay at the same position in the file then use `fflush()` or `fseek(fp,0L,SEEK_CUR)` which will move 0 bytes from the current position!

9.3 Program Arguments

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

`argc`	number of arguments
`argv`	array of strings (the arguments themselves)

For example, if your program is called `fred` and you type at the operating system prompt:

```
fred tom dick harry
```

then `argv[0]` will be the string `"fred"` (the program name), `argv[1]` will be `"tom"`, `argv[2]` will be `"dick"`, `argv[3]` will be `"harry"` and `argc` will be 4.

Some systems will set `argv[0]` to be the complete path to the executable program, and some set it to be a null string (i.e. a string containing only the `'\0'` character).

ANSI states that `argv[argc]` exists, and has the value `NULL`.

It is conventional to call the arguments to `main`, `argc` and `argv`, in fact any name can be used, but the types must be the same as shown above. Some systems also permit an optional third parameter, often called `env`, which is an array of strings holding the current set of environment variables. As there is no integer value which holds the number of elements of this array, the final `NULL` value held in the array must be used to determine the number of entries. The following example shows how to print both the `argv` and `env` arrays:

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

   for (i = 0; i < argc; i++)
      printf("%s\n", argv[i]);

   for (i = 0; env[i] != NULL; i++)
      printf("%s\n", env[i]);

   return 0;
   }
```

9.4 I/O Streams ‡

There are three standard I/O streams available for use; *standard input*, *standard output* and *standard error*. The identifiers `stdin`, `stdout` and `stderr` (defined in the header file `stdio.h`) refer to the three different streams. Normally `stdin` is connected to the keyboard and `stdout` & `stderr` are connected to the screen. All input routines refer to the `stdin` stream and all output routines refer to `stdout` unless otherwise directed.

9.5 Redirection of I/O Streams ‡

With many operating systems (including Unix and MS-DOS) it is possible (when running the program) to redirect the standard streams to other devices (e.g. a printer) or to a file instead of the screen or keyboard. With both Unix and MS-DOS redirection is obtained with the `<` and `>` operators. For example, if your program is called `fred`, then typing

```
fred
```

at the OS command line would connect `stdin` to the keyboard and `stdout` and `stderr` to the screen. If, instead, you typed (on an MS-DOS system):

```
fred < jim.dat > lpt1
```

then all input routines would read from the file `jim.dat` and all output routines would write to the parallel printer. MS-DOS does not permit redirection of the `stderr` stream, but Unix does; under the Bourne shell the following command would direct the standard output stream to a printer and standard error to the file `errors`.

```
fred < jim.dat > /dev/lp 2> errors
```

Unix and MS-DOS also permit *piping* of data from one program to another, thus

```
fred | jim
```

Here the `stdout` stream of `fred` is connected to the `stdin` stream of `jim` The same effect (though with the addition of the creation of a temporary file which would need to be deleted later) could be achieved by typing

```
fred > temp.tmp
jim < temp.tmp
```

The exact mechanism by which pipes are constructed should not interest us, but it is interesting to note that on a multi-tasking OS like Unix all the programs which make up a pipe are run concurrently.

It is considered good practice to write all error messages to the `stderr` stream so that any error messages stand a good chance of appearing on the screen, instead of disappearing into an output file or “down” a pipe.

10 Structures, Unions and Fields

10.1 Enumerated Types

```
enum day { sun, mon, tues, weds, thur, fri, sat } d1, d2;
enum suit { spades, hearts, clubs, diamonds }     s1;

enum suit    s 2, s3;
```

The identifiers in the enumerated type list can be used in assignments or tests, for example:

```
   d1 = thur;
   s1 = hearts;

   if (d1 == sat) ...

   switch (s1) {
      case spades :
      ...
```

Each value of the enumerated type list is given an `int` value. If no extra information is provided by the programmer, the values start at zero and increase by one from left to right. The value of any constant in the enumeration list can be set by the programmer; subsequent enumeration constants will be given values starting from this value. There is no need for the values given to the enumeration constants to be unique.

The following (non-sensical) code fragment illustrates these three points:

```
enum day { sun=1, mon, tues, weds, thur, fri=1, sat }  d1;
```

the values of the enumeration constants will be:

`sun` 1, `mon` 2, `tues` 3, `weds` 4, `thurs` 5, `fri` 1, `sat` 2

Enumeration constants *must* be unique across all enumerated types currently in scope; the following would be illegal:

```
enum day { sun, mon, tues, weds, thur, fri, sat }     d1;
enum weekend { sat, sun }                             d2;
```

The base type of an enumerated type is `int`, and values of type `int` can be assigned to variables of an enumerated type, although such use may be meaningless, e.g.

```
enum day { sun, mon, tues, weds, thur, fri, sat }     d1 = 76;
```

Some debuggers are able to show the values of variables of enumerated type using the identifiers used in the enumeration list, this helps with debugging.
As with structures and unions, declarations of enumerated types are local to the block in which the declaration occurs.

10.2 Defining New Names for Types with *typedef*

It is possible to create new names for existing types with `typedef`. This is frequently used to give shorter or less complicated names for types, making programming safer and hopefully easier.

For example, consider declaring a function parameter which is a pointer to a function which itself takes an array of strings as a parameter and returns a string:

```
typedef char *string;    /* string is a pointer to a char */
typedef string  strfn(string []);
                         /* strfn is a function that takes
                            an array of strings as a
                            parameter and returns a string */
typedef strfn   *pstrfn; /* pstrfn is a pointer to a  strfn */

void fred(pstrfn f)
   {
   }
```

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

Placement Classes

For All Companies placement Material

@placementclasses

CAT

For CAT Exam Preparation Material

@cat_classes

GATE

For GATE Exam Preparation Material

@gate_classes

EBOOK

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C Language

C

@c_examples

C++

@cpp_examples

Java

Java

@java_examples0

Python

@python_examples

Without using `typedef` the declaration would be somewhat more complicated:

```
void fred(char *(*fn)(char *[]))
   {
   }
```

`typedef` declarations are local to the block in which they appear.

10.3 Structures

Structures are variables that have several parts; each part of the object can have different types. Each part of the structure is called a *member* of the structure.

There are two ways to declare structured variables. You can declare a type of your own and use that type name to declare as many variables as you wish of that type, e.g.

```
struct date  {
             int day, month, year, yearday;
             char  monname[4];
             };

struct date  d;
struct date  d1 = {4, 7, 1776, 186, "Jul"};
```

Or you can declare the variables directly:

```
struct date  {
             int day, month, year, yearday;
             char  monname[4];
             } d, d1 = {4, 7, 1776, 186, "Jul"};
```

The two declarations of `d1` above show that initialisation of structures takes the same form as that of arrays.

The word following the keyword `struct` is the *tag* name of the structure, it gives the structure definition a type name that can be used in later declarations of variables, function parameters and return values. The tag identifier can be omitted if the type name is not needed for subsequent declarations.

Individual members of the structure are accessed by use of the `.` (pronounced 'dot') operator (as in Pascal). If the structure is accessed through a pointer, the `->` operator can be used.

```
struct date  {
             int day, month, year, yearday;
             char  monname[4];
             };

struct date  d1;
struct date  *d2 = malloc(sizeof(struct date));
```

```
d1.month = 7;

(*d2).year = 1993;
d2->year = 1993;
```

The last two statements show the alternative ways of accessing the structure through a pointer and they are exactly equivalent. The brackets are required to dereference the pointer `d2` before accessing the structure element.

K&R compilers will not permit structures to be passed by value to a function or to be returned from a function (pointers must be used instead in both circumstances), nor will they allow automatic structures to be initialised. No such restrictions exist in ANSI C.

Declarations of structures and unions obey the usual scoping rules, thus it is possible to localise a declaration of a structure to a block.

Whole structures cannot be compared directly, instead a member by member comparison is needed.

10.4 Unions ‡

A variable of union type may hold (at different times) objects of different types and sizes, the objects all occupying the same area of storage.

```
union tag   {
         members;
         } variables;
```

For example:

```
union value       {
         int intval;
         float fval;
         char  *pval;
         } uval;
```

The variable `uval` can hold three different types of object, an `int`, a `float` or a string. It is the responsibility of the programmer to ensure that they access the variable in the appropriate manner.

Access to the union members is via the `.` operator as with structures, e.g. the data in the union above may be accessed as

```
uval.intval
uval.fval
uval.pval
```

Also, as with structures, the tag identifier can be omitted if the type is not needed for further declarations.

When initialising a union, the initialiser must be a valid initialiser for the *first* member of the union enclosed within braces. It is only possible to initialise the `int` member of the example union above, e.g.

```
union value  u = { 6 };
```

10.5 Fields ‡

A field is a member of a structure whose bit length is specified. This can be useful for economic storage (not nearly as important as it used to be); but be warned that the code required to access bit fields becomes more complicated and so the program may have both a larger code size and also run more slowly.

```
struct   {
         unsigned  is_keyword   : 1;
         unsigned  is_extern    : 1;
         unsigned  is_static    : 1;
         } flags;
```

Fields are also used when access to individual bits of some part of memory is required. An example might be access to the status bits of an I/O port.

11 More Advanced Topics‡

11.1 Comma Operator

```
expr1, expr2
```

evaluate `expr1` then evaluate `expr2`. The complete expression has the value of `expr2`.

The three separate code fragments below are equivalent, but increasing use of the comma operator reduces the number of lines of code. This is not always a good thing!

```
sum = 0;
for (i = 1; i < n; i++)
   sum += i;

for (sum = 0, i = 1; i < n; i++)
   sum += i;

for (sum = 0, i = 1; i < n; sum += i, i++);
```

11.2 Conditional Operator

```
expr1 ? expr2 : expr3
```

Evaluate `expr1`

- if non-zero (true) evaluate `expr2`
- if zero (false) evaluate `expr3`

The value of the whole expression is the value of whichever of `expr2` or `expr3` was evaluated.

```
double maximum(double a, double b)
   {
   return (a > b) ? a : b;
   }
```

11.3 Name Spaces

C keeps four different *name spaces* for the following categories:

- objects (variables and function parameters), function names, `typedef` names and `enum` constants
- labels (for `goto`)
- `struct`, `union` and `enum` tags
- `struct` and `union` members (actually a different name space for *each* `struct` and `union`)

Identifiers from one name space will not clash with names from another. Thus, it is possible to have a function and a structure tag with the same name.

The following declaration is legal in ANSI C.

```
typedef struct fred   {
                      int fred;
                      } fred;
/* create a new type name fred, which is a structure whose
tag is fred, and which has an integer member called fred */
```

11.4 Type Qualifiers ‡

The qualifiers `const` and `volatile` can be added to the declaration of variables (or function parameters) to indicate:

`const`	the variable can be initialised, but cannot subsequently be modified
`volatile`	the variable might change its value at any time (an example might be a status register) and therefore it should not be optimised out of loops or expressions

`const` is preferred over the use of symbolic constants, as the constant object obeys the scope rules in the same way as variables. But note that `const` values cannot be used as a size specifier for an array, whilst a symbolic constant can.

```
#define N     10
const int M = 10;

int main()
   {
   int n[N];       /* OK */
   int m[M];       /* illegal */
   }
```

When `const` is applied to an array, it indicates that the array elements cannot be modified.

```
const double     pi = 4 * atan(1.0);
const char      usage[] = "USAGE: ";
```

With pointer objects (see section 8) there are three alternatives for using `const`:

- a constant pointer (Address to which it points cannot be modified)
- a pointer to a constant object (Objects value cannot be changed)
- a constant pointer to a constant object (Neither the address or the value can change)

Examples of these are given below:

```
char * const          p;
      /* p is a constant pointer to a char */
const char            *q;
      /* q is a pointer to constant char */
const char * const    r;
      /* r is a constant pointer to constant char */
```

11.5 Functions as Parameters

```
double sum(double (*f)(int), int m, int n)
```

The function `sum` has three parameters; the first is a pointer to a function that returns a `double` and takes an `int` as a parameter. One way of specifying such a pointer is by using the name of a function of the appropriate type. For example,

```
sum(xSquared, 0, 10);
```

where `xSquared` is a function which returns double. Within the function `sum`, the function that was passed in as a parameter can be called in either of the following two ways:

```
(*f)(m);
f(m);
```

The latter is not permitted in K&R C.

11.6 Preprocessor Macros

You can use the preprocessor to create powerful macros.

```
#define identifier replacement
```

For example:

```
#define SQ(x)      ( (x) * (x) )
#define MIN(x,y)   ( ((x) < (y)) ? (x) : (y) )
#define PI         3.14159
```

In effect you can write often used simple functions as a macro rather than a function, the advantage is speed (there is no function call overhead) and no type definition is necessary (it can be resused for `ints, doubles, floats` etc.); the disadvantages are that if you use the macro several times, then the code will appear in your program several times, and there can be side effects caused by not creating new copies of the parameters (which is what happens when a function is called).

When defining a macro there cannot be any spaces between the macro name and the parameter list; when calling a macro no such restriction holds.

Macros need not take parameters; these are sometimes called *symbolic constants*.

Why are so many brackets needed? The brackets are required to give the correct result when expressions are passed to the macro. For example:

```
#define SQ(x)   ( (x) * (x) )
#define sq(x)   x * x
```

```
SQ(a + b)       ⇒ ( (a + b) * (a + b) )
sq(a + b)       ⇒ a + b * a + b
```

The second macro gives the result `a + a*b + b` which is obviously not what was intended.

The outer set of parentheses are required to protect the whole expression from the vagaries of operator precedence:

```
1 / SQ(a)       ⇒ 1 / ( (a) * (a) )
1 / sq(a)       ⇒ 1 / a * a   =  a / a
```
(except when `a` is large)

The `SQ` macro still has one problem, which is shown by the following code:

```
SQ(a++)          ⇒ ( (a++) * (a++) )
```

The variable `a` ends up being incremented twice! This is certainly a surprise to the user of the macro. Watch out for any macro where the argument appears more than once in the replacement text.

11.7 Assertions

Assertions provide a good method of debugging. `assert` is a macro that tests to see if the value of the parameter is true or false. If the value is false, execution of the program is halted, and a suitable error message is printed onto the standard error stream.

Assertions are not available in K&R C.

The following example from [2] shows the use of assertions to verify that the initial conditions for a string copy operation are correct (`size_t` is the ANSI C type name for values holding size information, on the Borland Turbo C compiler, used in class, it is a synonym for `unsigned int`):

```
#include <assert.h>

char *CopySubStr(char *strTo, char *strFrom, size_t size)
{
   char  *strStart = strTo;

   assert(strTo != NULL && strFrom != NULL);
   assert(size <= strlen(strFrom));

   while (size-- > 0)
      *strTo++ = *strFrom++;
   *strTo = '\0';

   return (strStart);
}
```

If either assertion failed then an error message something like the following would be displayed:

```
Assertion failed: file C:\PROGRAMS\STRING.C, line 7
```

This may help you debug your programs more quickly. After you have finished debugging, do not remove the assertions from your code, instead add the following *before* the file `assert.h` is included:

```
#define NDEBUG
```

This will turn off all the assertions, but it is then very easy to turn them back on again. Alternatively if your compiler allows the specification of symbolic constants as a command line parameter when compiling, add `-DNDEBUG` to the other compiler flags.

Do not use assertions to deal with situations that can legitimately occur during the run of your program; for example failure to open files. The following example is *not* the way to use `fopen`:

```
int main()
   {
   FILE  *fp;
   char  filename[50];

   scanf("%49s",filename);
   fp = fopen(filename, "r");
   assert(fp != NULL);

   ...
   }
```

If the file cannot be opened the program will halt. It would be better to give the user the chance to enter the filename again.

12 Managing C programs

12.1 Separate Compilation

By making use of definition and declaration of functions we can split a program into sections in separate files. This allows us to manage large programs by breaking them down into smaller, more easily understood sections. It also makes testing easier (when a section is changed, you only have to retest that section and the files that depend on it) and encourages re-use of code.

When a program is compiled it only requires the declaration of a function (the function name, the number and type of parameters and the return type). This is enough to allow the calling code to be checked and compiled. The statements in the function (the definition) are only required when linking the program to form the excutable code.

However, the functions will need to be declared both in the file containing the functions and in the calling code to allow them to be compiled separately. It is not necessary to declare them twice though, a single header file is used for the declaration which then can be included (using a `#include <header.h>` statement) in as many other files as required. This also ensures consistency between declarations across the files.
In the following example three files are used:

Main program file	`main()` function calling library functions
Function Definition file	Code statements for the library functions (definition)
Header file	Declarations for the library functions

```
/* mainprog.c */
#include <funcdecl.h>          /* get function declarations*/
main()
   {
   function1();               /* Call function1 */
   function2("Fred");         /* Call function2 */
   return 0;
   }
```

```
/* function.c */

/* define the functions */

int function1(void)
   {
   printf("A function from a separate file\n");
   return 0;
   }

void function2(char *instring)
   {
   printf("Hello %s\n",instring);
   }

/* funcdecl.h */

/* Function declarations */

int function1(void);

void function2(char *instring);
```

12.2 Conditional Compilation

It is possible to make the compiler compile parts of a program only if certain conditions are met. This is frequently used where the same program is to be used with different C compilers (whether on the same computer or on a completely different system). For example, the "standard" library function to determine if a particular character exists in a string is `strchr` in Turbo C and on Unix C compilers, whilst the equivalent function is `index` in Zortech C. To produce a single source file which is compilable under all three systems, conditional compilation can be used.

```
#define UNIX          0  /* set to 1 the compiler used */
#define ZORTECH       0  /* all others should be */
#define TURBO      1  /* set to 0 */

#if    ZORTECH
   index(str, ch);
#else
   strchr(str, ch)
#endif
```

12.3 Using Projects in Borland C++

Projects are used in Borlands C/C++ compiler for DOS and Windows to manage programs that consist of multiple source files. A project file tracks not only which source and header files the project contains, but also which files depend on each other. When a file in a project is changed and the project recompiled, the compiler can recompile all the files that depend on that file, and thus check the effects of the change. Files that do not depend on the edited file do not need to be recompiled and this saves time when dealing with large projects.

The first step in creating a project is to create the source code and header files using an editor and save them. Then create a project by selecting `[Project/Open Project]` and type a filename. A project window will open. To add files to the project select `[Project/Add to Project]` and select the file required. The order in which files are added to a project is important. A file `B` which depends on another file `A` must be added after file `A` in the list. Repeat this step until all the files have been added to the project. The project file can be saved using `[Project/Save Project].` Once added to a project files can be edited by double clicking on their entry in the project window.

To create an executable file use `[Compile/Build All]` while in the project window. The project will be compiled. If a file has changed, all the files depending on that file will be recompiled automatically. Any errors will be reported and clicking on the message will open the appropriate file for editing. Use `[Project/Close Project]` to close the project file.

12.4 Unix and C

The command to compile a C source file under Unix will depend on what C compiler you have on your system. A typical example would be to use the `cc` command

```
cc program.c -o runfile
```

The `-o` flag allows the name of the executable file to be specified (the default is `a.out`). Make the `runfile` executable and run it using `./runfile`

Note: In order to use the `math.h` library under Unix the `-lm` flag should be used with the `cc` command.

In order to manage multiple files the `make` command is used to process a makefile. These work in the same way as project files in Borland C. The `make` facility is included in most C compilers , even those for DOS/Windows. The `make` command reads the names of files and the instructions for compiling and linking them from a file called a makefile. (See [9] for more information] or use `man make`).

To use the `make` facility under Unix create a file containing the following

- a list of all the the object files to be linked after the `run:` instruction.
- a list of compile commands and the object files to link.
- a list of the source files and header files.

Save it as `makefile` or another filename.

For example, to compile the files used in the example in the previous section the `makefile` should contain:

```
run: function.o mainprog.o
   cc function.o mainprog.o -o run
function.c mainprog.c : funcdecl.h
```

Use the command `make` to compile the file `makefile` or `make -f filename` if using a name other than makefile.

The executable file will be called`run`. (See appendix C, [6])

12.5 Header file locations

The location of the header file can be specified in the argument to the `#include` statement by using brackets or quotes.

`<` `>` Looks for header fille in default path specified by your system.
`"` `"` Looks in location specified in the quotes
Examples
`"funcdecl.h"` Current working directory
`"c:\cprog\funcdecl.h"` Specifies a directory in DOS or windows

13 Memory Usage ‡

When a C program is loaded into memory, the program instructions (code) and the external variables are usually placed in different areas in memory. The part of memory where the code exists is called the *text area*, and the memory where the external variables reside is called the *data area*. In addition automatic variables are allocated space on the *stack*, and dynamic memory is allocated on the *heap*.

13.1 Text Area

This is the memory used for the executable instructions of the program. This memory is usually *read-only*, which means that programs cannot modify their own code. Pointers to this region are permitted, in the form of pointers to functions, e.g.

```
int fred();

int main()
   {
   int   (*funcPtr)() = fred;
      /* funcPtr now points to fred() */

   (*funcPtr)();
      /* call the function that funcPtr points to */

   return 0;
   }
```

The text area can sometimes be shared between several versions of the same program, which will help save memory. With the Unix operating system, shared text is the default for every program.

13.2 Data Area

The data area is used for *external static*, *external* and *internal static* variables. External variables are those defined outside of any function. Internal variables are those defined inside functions. Static variables are those declared to be `static`. Internal variables are of class `auto` by default.

```
int          fred;        /* external */
static int   jim;         /* external static */

void fred()
   {
   float           x, y; /* internal automatic */
   static double   z;    /* internal static */
   }
```

13.3 The Stack

The stack is used for the values of actual function parameters, function return addresses and internal automatic variables. When a function returns the stack space used is marked as being free, but is not necessarily cleared, which can sometimes cause confusion, e.g.

```
void fred()
   {
   int   a;

   a = 15;
   }

void jim()
   {
   int b;

   printf("%d\n", b);
   }

int main()
   {
   fred();
   jim();
   return 0;
   }
```

The `printf` call in `jim` will probably print `15` as that will have been the value left in that memory location by the internal variable `a` in function `fred`.

13.4 The Heap

The heap is used to store dynamically objects in memory during the life of the program. It is up to the programmer to allocate storage on the heap for data, and to relinquish that space when it is no longer required. Space allocated on the heap remains in use until the end of the program or until freed with `free()`.

The heap is most often used to store dynamic objects which grow (and shrink) during the life of the program, such as linked lists and binary trees.

The library functions `malloc()`, `calloc()` and `free()` are used to allocate space on the heap and to free it. `calloc()` fills the allocated memory space with zeros, `malloc()` leaves it uninitialised.

A common problem with the heap is to allocate space but then to lose the pointer which gives access to that part of memory.

```
void fred()
   {
   char  *ptr;

   ptr = malloc(100);
      /* allocate enough room store 100 chars */
   scanf("%s", ptr);
      /* read in a string and store it on the heap */
   }
   /* OOPS, just lost our pointer to the data, as ptr was
      stored on the stack. The string is still on the heap
      but we have no means of accessing it! */
```

Another common mistake is to forget to free heap memory. In fact in the example above it is quite likely that the programmer intended the string to act as a local variable, in which case the statement

```
free(ptr);
```

should be added at the end of the function.

13.5 Possible Problems

Automatic variables are *not* implicitly initialised, which would involve a run-time overhead. External and static variables are initialised by the loader before the program starts running.

Not all compilers place constant strings in the data area, some compilers place constant strings into the text area which would make them unmodifiable (which ANSI C says they must be anyway).

Duplicated constant strings are probably duplicated in memory, but most compilers give the option to merge duplicated strings.

External and static variables cannot be initialised with values that are not known until run time. This is because the data area is filled by the loader *before* the program starts running.

```
void fred(char *msg)
   {
   int          l1 = strlen(msg);     /* legal */
   static int   l2 = strlen(msg);     /* illegal */
   }
```

Note that `sizeof` is an operator and can therefore be used to initialise any class of variable.

14 C and the IBM PC ‡

14.1 Memory Organisation

The IBM PC and compatibles all use the Intel 8086 family of processor. IBM XTs use the *8086*, IBM ATs use the *80286* and PS2s use the *80386*. More recent IBM PC (and compatible) computers use the *80486* and *Pentium* processors. Many of the features of the five processors are the same, but the 80286, 80386, 80486 and Pentium offer superior memory access modes over the 8086 (but they are compatible with the 8086 when they are put into *real* mode).

The 8086 family (with the exception of the Pentium) all use *segmented* memory when using *real* mode. The memory of the computer is divided into 64k (64 * 1024) byte chunks. Each of these 64k byte blocks is called a *segment*. To access any given memory location in your computer's memory you need to specify the address of the segment and then the offset of the memory location within that given segment. The designers of the 8086 decided that the way the segment address and offset address should be combined to produce a physical memory address should be:

```
segment << 4 + offset
```

As the segment address and offset are specified by 16 bit integers, the resulting physical address is 20 bits, which gives a range of addresses of exactly 1M (1024 * 1024) bytes. This is the maximum amount of memory that the 8086 can access. Some IBM PCs do have more than 1MB of memory, but some “trick” must be employed to make it look as though there is still only 1MB of “real” memory. One technique employed is to swap segments of memory to/from any memory over the 1MB limit into the real 1MB address space. An example of this is the LIM (Lotus Intel Microsoft) expanded memory system. Pentium processors use 32 bit Memory adressing, giving up to 4GB of address space.
Most of this is of little interest to C programmers until they start writing large programs. By large, I mean that either the amount of C code used is large or that the total memory size of variables used is large (or both). Then you need to start thinking about which *memory model* you should employ for any given C program.

There are seven different memory models (but not all are available with some compilers):

Model	Code size	Data size
tiny	64kB for code and data together	
small	64kB	64kB
medium	1MB	64kB
compact	64kB	1MB
large	1MB for code and data together (using unnormalised pointers)	
huge	1MB for code and data together (using normalised pointers)	
flat	4GB for code and data together	

The flat memory model (with 4GB of data space) is available on the 80386 and higher processors (including Pentium processors). Some DOS compilers are not able to use the flat memory model - one that can is D.J. Delorie's port of the GNU C compiler, *gcc*. Windows based compilers will have an option allowing you to select the memory model required.

The default memory model for most MS-DOS C compilers is small. This is adequate for most programs. However, you may find if you use large arrays in your program that the compiler or linker will warn you that your total variable storage exceeds 64kB. You should then recompile using the compact or large memory model. Changing from one model to another is simply done by menu options Borland C or Zortech C, or by command line arguments with command line compilers; see your compiler's documentation for exact details.

With the tiny, small and medium models *near* pointers (2 bytes) are used to address variables. Near pointers store *only* the offset within the data segment. As there is only one data segment this is all the information necessary. If the compact, large or huge memory models are used then *far* pointers (4 bytes) are used to access variables. The use of far pointers increases the size of the executable program and increases the time taken to access any variable. Similarly, the tiny, small and compact models use *near* pointers for function calls, whilst the medium, large and huge models use *far* pointers. This introduces a similar overhead on function calls. Therefore, if possible, use the small memory model.

14.2 BIOS (Basic I/O System) Interrupts

All MS-DOS computers have some basic routines stored in read-only memory in the computer. These routines provide a mechanism for communicating with the keyboard and screen, amongst others, and hence this part of memory is given the title *BIOS* (Basic Input/Output System). The BIOS can be thought of as coming between our C programs and the hardware of the computer itself. When we use a `printf` statement in our program, the compiler produces machine code which calls a subroutine in the BIOS to make characters appear on the computer screen. So generally we need not trouble ourselves with trying to understand exactly how the BIOS works.

However, the BIOS provides many routines that your C compiler may not support. Examples include communicating with serial/parallel ports, video card control (scrolling, writing, reading etc.), disk access, keyboard access (read key states, is a key waiting to be read, changing auto-repeat delay etc.) and many others. Remember C is supposed to be a portable language, and the examples given above are all specific to the IBM PC and the 8086 processor. Some versions of C do provide access to some of these routines via library functions.

All MS-DOS C compilers do provide a mechanism for calling BIOS routines directly though. Armed with a good guide to the BIOS routines (like Peter Norton's *Programmers Guide To The IBM PC* [4]) we can make our PC sing and dance at lightning speed.

I give some simple examples below, but for more information I suggest you consult [4].

The program below will move the cursor to any valid screen location - any subsequent screen output will start at this point.

```
#include <dos.h>

void GotoXY(int x, int y)
/* positions cursor at line y, column x */
   {
   union REGS  regs;

   regs.h.ah = 2;
   regs.h.dh = y;
   regs.h.dl = x;
   regs.h.bh = 0;
   int 6(0x10, &regs, &regs);
   }
```

The `int86` function copies the variable `regs` into the processor registers, executes the BIOS interrupt (routine) specified – in this case `0x10` (the video interrupt) - and then copies the processor registers into `regs` again. Modifying `regs` does not change the processor registers directly.

The following program will set the attribute bits of a given file.

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

int attrib(char *, int);

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

   if (argc != 3)
      {
      fprintf(stderr, "USAGE:     %s attributes filename\n",
argv[0]);
      fprintf(stderr, "       eg: %s 33 fred.c\n", argv[0]);
      fprintf(stderr, "           would set archive & read-
only ");
      fprintf(stderr, "           attributes of fred.c\n");
      return 1;
      }

   if ((attributes = atoi(argv[1])) > 0xFF)
         /* atoi converts an ASCII string to an int */
      {
      fprintf(stderr, "ERROR: invalid attributes %d\n",
attributes);
      return 2;
      }

   switch (attrib(argv[2],attributes))
      {
      case 0 :  /* OK */
                break;
      case 2 :  fprintf(stderr,"ERROR: file %s not
found\n",argv[2]);
                return 3;
      case 3 :  fprintf(stde rr,"ERROR: path %s not
found\n",argv[2]);
                return 4;
      case 5 :  fprintf(stderr,"ERROR: access denied on file
%s\n",
                                  argv[2]);
                return 5;
      }
   return 0;
   }

int attrib(char *filename,int attributes)
   {
   union REGS   regs;

   regs.x.dx = (int) filename;
   regs.x.cx = attributes;
   regs.h.al = 1;
   regs.h.ah = 67;
   int86(0x21, &regs, &regs);
   return regs.x.ax;
   }
```

No real damage can be achieved with this program, except to hide files without realising it (these can be discovered with a small `findfirst`/`findnext` program or with `dir/ah` in DOS 5 or higher).

14.3 DOS Interrupts

The MS-DOS operating system provides a great many subroutines for disk handling, video card manipulation, keyboard access, file operations, time & date functions and so on. These routines are generally at a higher level than the BIOS routines, but they are invoked in a similar manner.

One visual representation of the inter-relation of the computer, the BIOS, DOS and our C program is to think of the whole system as an onion, viz.

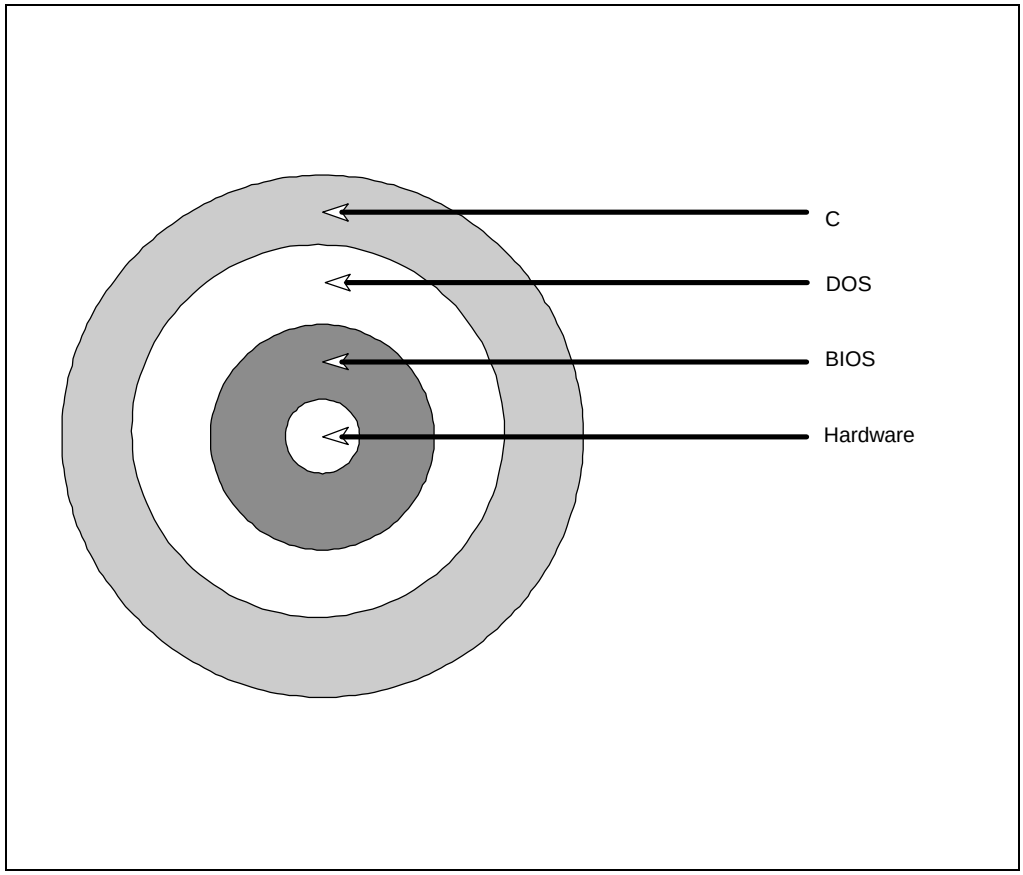


Figure 1 Layering of PC system architecture

The nearer the outside of the onion, the more machine *in*dependent our code should be, the nearer the centre of the onion the more *de*pendent our code is upon the computer we are using. Thus we would expect `printf` to work satisfactorily on most machines with a C compiler. We would hope that DOS interrupts will work in the same way on all computers with the same version of MS-DOS, whereas BIOS interrupts will only work correctly on computers with an IBM compatible BIOS system.

The following program shows the use of DOS interrupts to find files on a DOS disk.

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

char *farstrcpy(char *dst, char far *src)
   {
   char        *str = dst;

   while (*dst++ = *src++)
      ;
   return str;
   }

int main()
   {
   union REGS  inregs, outregs;
   char        str[13];
   char        *filespec = "c:*.c";

   inregs.h.ah = 78;  /* 78 = find first function */
   inregs.x.dx = (unsigned)filespec;
                     /* file specification */
   inregs.x.cx = 0;   /* file attributes */

   int os(&inregs,&outregs);
                     /* call DOS "find first" interrupt */

   while (outregs.x.cflag == 0)
      {
      printf("Found: ");

      farstrcpy(str,MK_FP(_psp,128+30));
      /* find first writes the matching file name 30 bytes
         into the "disk transfer area", which itself is 128
         bytes into the "program segment prefix" or PSP */

      printf("%s\n", str);

      inregs.h.ah = 79;
                     /* 79 = find next func tion */
      intdos(&inregs,&outregs);
                     /* call DOS "find next" interrupt */
      }

   return 0;
   }
```

Borland C compilers provide two library functions `findfirst()` and `findnext()` to find files matching a given path name and attribute, but these functions are merely "front-ends" to the two DOS interrupts.

`findfirst()` initialises the search and fills in a structure giving details of the first matching file (if there is one). Repeated calls to `findnext()` will find all remaining matching files. Both functions return `-1` if no match is found.

```
int findfirst(const char *pathname,
          struct ffblk *ffblk, int attrib);
```

```
int findnext(struct ffblk *ffblk);
```

The prototypes for `findfirst()` and `findnext()` are in `dir.h`.

The structure called `ffblk` is defined in `dir.h` to be

```
struct ffblk    /* struct used by findfirst() and findnext()
*/
     {
        char  ff_reserved[21];   /* reserved by DOS */
        char  ff_attrib;         /* attribute found */
        int   ff_ftime;          /* file time */
        int   ff_fdate;          /* file date */
        long  ff_fsize;          /* file size */
        char  ff_name;           /* found file name */
     };
```

The file attributes are set by MS-DOS (they are nothing to do with C). They are

Bit	**Hex**	**Indicates**
0	0x01	read only
1	0x02	hidden
2	0x04	system
3	0x08	volume label
4	0x10	sub-directory
5	0x20	archive bit
6	0x40	unused
7	0x80	unused

Each individual attribute is set by setting a particular bit of an 8 bit byte.

The rules for finding files with specific attributes are rather strange, again this is a "feature" of MS-DOS and has nothing to do with the functions provided by Borland or other compiler manufacturers:

> "Any combination of the hidden, system or directory attribute bits will match *normal* files as well as those with the attributes given. If a volume label attribute is used, then *only* the disk label will match. The archive and read-only bits do not apply to the search."

To find files with combinations of attributes the separate attributes must be bitwise **OR**ed together.

The following program will print the names of all C source files in the current directory.

```
/* findfirst and findnext example */

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

```
int main(void)
   {
   struct ffblk    ffblk;
   int             done;

   printf("Directory listing of *.*\n");
   done = findfirst("*.*",&ffblk,0);
   while (!done)
      {
      printf("  %s\n", ffblk.ff_name);
      done = findnext(&ffblk);
      }

   return 0;
   }
```

It is usually best to use the highest level interface available to a given operation, that is:

ANSI C
Vendor specific C function
DOS interrupt
BIOS interrupt

The higher the level, the more protected you are from specific models of the IBM PC, specific versions of MS-DOS and specific compilers. Choose an ANSI C function if at all possible for the most portable solution.

14.4 Dynamic Link Libraries (DLLs)

A DLL is a file containing compiled functions which can be linked into your program at run time. This makes your program smaller but does require the correct DLLs to be available when your program runs. The declarations for functions in the common DLLs are usually supplied in a text file, and the appropriate declarations can be copied into your code as required. The Windows operating system and applications written for it make extensive use of DLL's for common functions.

Some compilers allow you to compile your code to a DLL file. The source code language is not important, however variables and functions that are going to be used from outside the DLL must have the correct declarations. Consult your compiler documentation for further details on how to do this.

14.5 Windows Application Programming Interface (API)

Windows is an operating system made up from a set of functions, messages, data structures, data types and statements that work together to create applications that run under the Windows operating system. The functions that make up the Windows API can be used from within your own C programs. This involves copying the declarations for the data and functions from the API into your own code before you can use them. The declarations are usually held in text files. For more information see the help system for your compiler or the Microsoft Software Developers Kit (SDK) [7].

15 Why C?

15.1 Evolution of the C Language

The C Programming language was written by Dennis Ritchie of Bell Laboratories. The first C compiler ran on a DEC PDP-11 computer running the Unix operating system. Later versions of Unix were written almost entirely in C! The language was designed by Ritchie to help him with his work on designing and implementing the *Unix* operating system.

C is based on the languages *BCPL* and *B*, hence the name *C*. B was written in 1970 by Ken Thompson for the first version of Unix, which ran on the PDP -7 computer. Many of the characteristics of B and BCPL are found in C, but C differs from these two languages in one very important respect. BCPL and B are *typeless* languages, the only data type is the machine word. This data type is used directly or indirectly (via special operators) to store all the data used in any program. In C there are several "built-in" data types (e.g. `char, int, float, double`) and any number of derived data types (arrays, structures etc.).

C is a relatively "low level" language in that C has few operators which act on collections of data such as arrays, strings and sets. Neither does the C programming language have any way of communicating with the outside world - there are no input or output routines in C. These operations are provided by a collection of standard library functions which are available with every implementation of C (i.e. C compiler).

15.2 C and Operating Systems (esp. Unix)

As mentioned above, the first C compiler was written for the Unix operating system. The language was then used to write many of the plethora of Unix utilities. The logical step, then, was to write the next version of Unix in C itself. This set the trend in operating system development. Being able to write an operating system in a high level language has obvious benefits in terms of development time, bug detection and correction, and maintenance of the software. All versions of Unix since the development of C have been written in C. Most versions of MS-DOS have been written in C. Numerous other operating systems and environments have been developed in C, even if parts are later converted to assembler. Examples include Microsoft Windows, Atari TOS, Amiga OS etc.

Programmers writing systems programs in C have the entire computer under their control. There is not one part of the computer memory they cannot access. All the bit manipulation routines are available. Control over memory organisation can easily be achieved. And yet they are writing in a high level language with all the benefits that HLLs bring, like type checking, loops, branching, variables, arrays etc.

Goodbye assembler - hello C!

15.3 Comparison with Pascal And Fortran

Although C is not a typeless language as BCPL was, it does not have the strict type checking that Pascal (or Algol 68) has. You will find as you write your own C programs that the C compiler will not always inform you about any mistakes you have made regarding type mismatches.

There is no run-time checking of array subscripts and with non-ANSI C compilers there is no verification of function argument types as there is in Pascal and Fortran.

Many Unix systems have `lint` a program that is designed to check C source files removing the "fluff"such as errors, non-portable statements and inefficient code. `lint` is also available (commercially) for other operating systems. `lint` answers many of the criticisms of C compilers given above, but `lint` is not always available.
Otherwise there are many similarities between C and Pascal & FORTRAN. Of course there are differences: C and Pascal permit recursion whereas FORTRAN (pre FORTRAN 90) does not; C and FORTRAN permit modular compilation whereas standard Pascal does not (some, notably Turbo Pascal, do); C and Pascal have pointer types whereas FORTRAN does not (FORTRAN 90 does!); parameters to C functions cannot be passed by reference as they can with Pascal and must be with FORTRAN.

Overall C stands for speed and power. But it is not for the uninitiated. It can be a very unforgiving language. The compiler believes that you know what you are doing; you are in control! It's a bit like riding a bike, terrifying when you are being pushed along at the start, and you may fall off a few times. But once you have found your balance and plucked up your courage you will find it much quicker to get to your destination!

15.4 Availability

The C language is almost certainly available on more machines then any other programming language. The fact that C is used to provide the operating system and system tools for most new computers means (ipso facto) that a C compiler for that machine has been developed (actually it doesn't — but it usually does).

15.5 Portability

With care your C programs can be written in such a way that the programs can be easily transferred from one C compiler to another. One problem when porting programs to a new compiler or new machine are differences in the "standard" library routines provided with the compiler. ANSI C does now specify a standard library that *all* ANSI compilers must provide.

15.6 Efficiency

Efficiency has many meanings. It could mean any of the following:
- ability to produce programs which run quickly
- production of small executable programs
- being able to write a given algorithm in fewer program statements

15.6.1 Speed

As a direct result of the first C compiler being produced for the PDP -11 computer, many of the C operators (especially increment & decrement) convert to single instructions in PDP-11 assembler. This meant that some C programs were faster than equivalent Fortran programs on the PDP -11. To a certain extent this still holds, some operations still match well with most processors. However, a good Fortran compiler may well produce faster code than a C compiler, it really all depends on the quality of the compiler.

15.6.2 Size of Executable File

The calling mechanism that C uses for functions produces smaller and less complex machine code than the Pascal or Fortran system does. Thus the overhead in calling functions is reduced. Also, as described above, many C operators translate to single machine instructions. C programs are often smaller than Pascal or Fortran equivalents.

15.6.3 Size of Source Code

C has many tricks and short-cuts which allow the programmer to write in a couple of lines something that would require 10 or more lines of Pascal. Whether this is of any real advantage is debatable, as updating the program, either at a later date by the same programmer or by another programmer entirely, becomes progressively more difficult as more of the shortcuts are used. We will come to something of a compromise between shorthand methods and *readable* and *maintainable* code.

15.7 Modular Programming and Libraries

C provides the ability to split large programming tasks into separate *modules*. Each module can consist of a C source file which can be compiled separately from all the other modules. With careful use of static external variables, a certain amount of information hiding can be performed - but not in anything like the secure methods of Modula-2, Ada or C++. To produce a single executable program the separately compiled modules are linked together, along with any library routines used, by a special program called (surprisingly enough) a *linker*. Fortran supports modular compilation, ISO standard Pascal does not. For almost all real applications this feature is a *must*.

15.8 Applications

As we have already seen, C is the *de facto* standard for systems programming. Virtually all new operating systems are written in C. C is also used for many commercial programs where speed of program execution or the time taken to produce the program is of vital importance. Examples include operating systems (Windows), applications programs (wordprocessors, databases and spreadsheets) and so on. In the good old days, parts or all of such programs would be written in assembler (the original version of the *WordStar* word processor was written entirely in assembler). Now such programs are written in C, which means that development time (and therefore *cost*) is significantly reduced.

15.9 Kernighan & Ritchie C vs. ANSI C

Dennis Ritchie, the creator of C, is also the author (with Brian Kernighan) of, possibly, the most important C text book *The C Programming Language* [1]. As there was not an international standard for C, the appendix of the first edition of this book (*C Reference Manual*) formed the *de facto* C "standard". Compiler writers were not obliged to keep to the behaviour specified in the book, but in the main they did. This informal C standard is called *Kernighan and Ritchie C* or *K&R C*.

In 1989 the ANSI C standard was finalised, and shortly after became an ISO (International Standards Organisation) international standard. C that complies with the standard is called *ANSI C*. ANSI C offers many useful extensions to K&R C and tightens up some loopholes. ANSI C is superior to K&R C, especially in the area of function declarations and prototypes.

15.10 Criticisms of C

Some of my own *personal* criticisms of the language are that C

- overuses operators
- allows misuse of types
- can be difficult to read
- can be good at (temporarily) killing PCs!

although some of these are addressed, partially at least, by ANSI C.

16 Other Courses of Interest

Object Oriented Programming in C++ introduces the differences between C and C++, and the concepts and syntax of classes and objects. Most programs are written using Objects and knowledge of C++ is a widely sought after skill in the programming market.

17 Bibliography

The recommended book for the course is:

- *Programming with ANSI C*, Brian J. Holmes, DP Publications 1995.

For further reading, I suggest the following:

- The C Programming Language, Brian Kernighan & Dennis Ritchie, Prentice Hall, Second Edition, 1988.
- C by Dissection, Al Kelley & Ira Pohl, Benjamin/Cummings, Second Edition, 1992
- C Traps and Pitfalls, Andrew Koenig, Addison-Wesley, 1989
- Advanced C, Tips and Techniques, Paul Anderson & Gail Anderson, Hayden, 1988
- Writing Solid Code, Steve Maguire, Microsoft Press, 1993

The C Programming Language, Second Edition (often referred to as K&R2) is an excellent reference on ANSI C. It is clear and succinct. If you only want to buy one C book, and you want it to last you a lifetime, get this one. However, K&R2 is not for beginners, it is most useful if you have experience (perhaps a lot!) of programming in another language.

18 Exercises

Example solutions to these exercises can be found in the OUCS guide l9.5 Programming in C - Exercise solutions [10].

Exercise 1 - "Hello world"

Type in the following program

```
#include <stdio.h>

int main()
   {
   printf("Hello World\n");
   printf("Hello World\n");
   return 0;
   }
```

Then compile and run the program.

Exercise 2 - Run-Time Error

The following program has a run-time error in it.

```
int main()
   {
   int      x, y = 0;
   x = 1 / y;

   return 0;
   }
```

Run the program to see what the effect of the error is. Rewrite the program without the variable `y`, but keeping the error in the program. What happens when you try to compile the program?

Exercise 3 - Increment (++) Operator

Write a program containingthe line

```
a -= a++;
```

What happens to the value of `a`?

Exercise 4 - `printf()` and `stdio.h`

Write a C program that prints on the screen the words

```
Now is the time for all good men to come to   the aid of their
country
```

- all on one line;
- on three lines;
- inside a box composed of `*` characters.

Exercise 5 - `printf()` Formats, Mixed Mode Arithmetic

The following program contains three statements which either lead to error messages or print unpredictable values. Correct them by changing the `printf` conversion characters and try to predict the output of the revised program. Run the program to verify your predictions.

```
#include <stdio.h>

int main()
   {
   int      i = 2;

   printf("%d\n", 8*i/3);
   printf("%d\n", 8.0*i);
   printf("%f\n", 8*i);
   printf("%f\n", 8*i/3);
   printf("%d\n", 8*i%3);
   printf("%f\n", 8.0*i/3);
   return 0;
   }
```

Exercise 6 - Using `printf()` Format Specifiers

Write a test program to find out whether the `printf` function truncates or rounds when printing out a float with a fractional part.

Exercise 7 - Using `for` Loops

Write a program that prints a table of powers. The first few lines might look like this

```
integer     square      cube     quartic     quintic
-------------------------------------------------------
      0          0         0           0           0
      1          1         1           1           1
      2          4         8          16          32
      3          9        27          81         243
      4         16        64         256        1024
```

Exercise 8 - `scanf()`, Printing Numbers In Different Bases

Write a program that reads in an integer and prints out the given integer in decimal, octal and hexadecimal formats.

Exercise 9 - `if ... else` Decisions

What gets printed?

```
#include <stdio.h>

int main()
   {
   int      i, j;

   i = j = 2;
   if (i == 1)
      if (j == 2)
         printf("%d\n", i = i + j);
   else
      printf("%d\n", i = i - j);
   printf("\n%d", i);

   return 0;
   }
```

Try it on paper first and then verify your answer with the computer.

Exercise 10 - `while` and `for` Loops

Rewrite (on paper) the following code as a single `for` loop with a body that consists of a simple (not a compound) statement.

```
i = 0;
while (c != ' ' && c != '\n' && c != '\t' && c != EOF)
   {
   word[i++] = c;
   c = getchar();
   }
```

Exercise 11 - `unsigned int` Type

Explain the output of the following program. What is the effect of changing the declaration of `x` from `unsigned` to `int`? What is the effect of changing the format from `%u` to `%d`.

```
/* compute powers of two */

#include <stdio.h>

int main()
   {
   int      i = 0;
   unsigned x = 1;

   while (i < 35)
      {
      printf("%8d:%8u\n", i, x);
      /* %u is unsigned decimal format */
      ++i;
      x *= 2;
      }
   x = -1;
   printf("\n\nminus one = %8d%8u\n\n", x, x);

   return 0;
   }
```

Exercise 12 - `getchar()`, `putchar()` and `if` Statements

Write a program to check the proper pairing of braces. The program will have two variables, one to keep track of the left braces and the other to keep track of the right braces. They both start at value 0 and the appropriate one is incremented each time a brace is encountered. If the right brace variable ever exceeds the value of the left brace variable, the program inserts the character pair `??` at that point in the output. If, at the end of the input file, the left brace variable is greater than the right brace variable, the program should print a message that include the number of missing right braces as a series of that many `}`s. Use `getchar()` and `putchar()` to do the input/output. What output is produced for the input `}{}{`? Do you think that the output is correct? Try to rectify the algorithm so that the output is `}??{}{ 1 missing }` .

Exercise 13 - `getchar()` and `EOF`

Write a program that counts the number of blanks, tabs and newlines that are typed on the keyboard. The program should terminate when the `EOF` character is read. The tab character is `'\t'`.

Exercise 14 - User Defined Functions and `scanf()`

Write a function `FtoC(double f)` which converts the Fahrenheit temperature *f* into the equivalent Celsius value. The function should return a value of type `double`. Use the formula:

c = 5/9 * (f - 32)

Test your function with the temperatures 32 °F (0°C), 212°F (100°C) and -40°F (-40°C).

Exercise 15 - User Defined Funtions, Prototypes and `math.h`

Write a function `power(x, n)` that will compute the n^{th} power of *x*, where *x* is a `double` and *n* is an `int`. As the power will be an integer, repeated multiplication using a `for` loop is one solution. Your function should return 1 when *n* is 0. Extend your function to deal with negative *n*.

Exercise 16 - Passing Parameters to Functions By Reference

Write a function called `swap` that will exchange the two `double` parameters given. Test the function by assigning two different values to two variables and using the function to swap the two values. Print out the values of the two variables after the function call to check that the function has performed the correct action.

Swapping is used in many applications, but particularly in sorting. If you feel adventurous write a bubble sort program.

Exercise 17 - `#define`

Andrew Koenig states in *C Traps and Pitfalls* [3] states that the following causes an infinite loop on some systems. Can you work out why?

```
#define  N  10

int main()
   {
   int   i;
   int   a[N];

   for (i = 0; i <= N; i++)
      a[i] = 0;

   return 0;
   }
```

Exercise 18 - `register` Variables, `rand()` and `stdlib.h`

Write a program to store 1000 random numbers in an array. Use an integer as a loop counter and to access the array. Print out the array. Time the program.

Try the program again but with the loop counter/array element variable as a register variable. Time the program again.

Was there any difference in the timings?

Exercise 19 - Pointer Variables, *(indirection) and &(address) Operators Strings: Arrays of Characters

Write a program with the declarations

```
char  a, b, c, *p, *q, *r;
```

and print out the locations that are assigned to all these variables by your compiler (that is print the values of the expressions `&a`, `&b`, `&c`, `&p`, `&q` and `&r`; use `%u` as the conversion specifier for each expression).

What can you deduce about the sizes of `char` and pointer variables on your system?

Add the following statements to your program:

```
a = 'A';
b = 'B';
c = 'C';
p = "hello";
q = "world";
```

and print out the values of the following expressions (the appropriate conversion specifier is given alongside each expression):

```
a     %c
b     %c
c     %c
*p    %c
*q    %c
p     %s
q     %s
```

Exercise 20 - Pointers to `NULL`

What is printed and why?

```
#include <stdio.h>

int main()
   {
   char     *pc = 0;
   int      *pi = 0;
   double   *pd = 0;

   printf("%8d%8d%8d\n%8d%8d%8d\n",
            pc + 1, pi + 1, pd + 1, pc + 3, pi + 5, pd + 7);

   return 0;
   }
```

Exercise 21 - Strings: Arrays of Characters

Write a program which reads a word and prints the word reversed. Use an array of characters to store the word read.

FREE EBOOKS, NOTES , VIDEOS & PLACEMENT MATERIAL

Placement Classes

For All Companies placement Material

@placementclasses

CAT

For CAT Exam Preparation Material

@cat_classes

GATE

For GATE Exam Preparation Material

@gate_classes

EBOOK

For Engineering Books & Material

@cs_ebooks

Codes of Following Programming Languages

C Language

C

@c_examples

C++

@cpp_examples

Java

Java

@java_examples0

Python

@python_examples

Exercise 22 - String Functions in `string.h`: `gets()` and `strlen()`

Write a function called `strlen` which will return the length of a string.

Exercise 23 - String Functions: `strcmp()`, `stricmp()`, `toupper()`

The C library function `strcmp` compares two strings. It is case sensitive. Write a function called `striccmp` (string ignore case compare) which will compare two strings ignoring case. The function should return `0` if the strings are the same, otherwise the difference in value between the first two non-matching characters.

```
striccmp("Word", "wOrd")        would return 0
striccmp("steve", "steven ")    would return -110
striccmp("box", "book")         would return 9
```

You may like to use the C library function `toupper` in your compare function. `toupper` converts its int parameter to upper case. It returns an `int` value.

```
toupper('c')                    returns 'C'
toupper('C')                    returns 'C'
```

[*Warning:* some older C libraries only guarantee that `toupper(`*`character`*`)` works if the argument is a lower case letter.]

Finish off by writing a program that tests the function in several different ways. This need only be a group of function calls printing the result returned by the function.

Exercise 24 - User Defined String Functions

Write a function called `strchr` which finds the first occurrence of a specific character in a given string. The function should return a pointer to the occurrence of the given character, or `NULL` if the character is not found.

```
strchr("hello world", 'l')      would return "llo world"
strchr("C programming", 'z')    would return NULL
```

Test your function by calling it several times with different literal strings (as shown above) and printing the result returned by the function. Unexpected results can occur if you try to print out a string which is actually a `NULL` pointer, so you will need to test that function did not return `NULL` before you try to print the string.

Exercise 25 - User Defined String Functions, `atoi()`

The parameters passed to a program are always strings. Sometimes you may wish to write a program that takes as its parameter an integer value, e.g. `add 6 2` would call a program called add which would add up all its parameters and print the result on the screen.

Write a function which takes a string (pointer to `char`) as a parameter and returns the integer equivalent.

`str2int("1658")` would return the value `1658`

To improve your program change the function to accept strings with a leading - sign to indicate a negative number.

One last improvement might be to print a warning if the string represents a number that is outside the range of an `int`.

The ANSI C function `atoi()` performs this task, do not use it to complete the exercise!

Exercise 26 - `getc()` and `putc()`

Write `getstring()` and `putstring()` routines. They should use `getc()` and `putc()` to read and write strings from a designated file.

Exercise 27 - Input from Files

The standard way to list a file on the screen with MS-DOS is to type at the keyboard:

```
type filename
```

This can be extremely annoying as the text shoots past your eyes so fast that it is impossible to read it!

Write a program called `more` that asks for a name of a file to list on the screen and then displays the file on the screen. Use `getc()` to read the characters in turn from the file and then display the characters on the screen using `putchar()`. Remember to use an `int` to store the character read, as you will need to test the character against the `EOF` character.

Modify your program so that it displays the first 24 lines only of the screen on the screen. It should then display a message like

```
-- More --
```

and wait for any key to be pressed. Once a key has been pressed the program should display the next 24 lines of the file, and so on. You might like to use the `getch()` function which “bypasses” the keyboard buffer. `getch()` is unique to MS-DOS compilers, it is not part of ANSI C.

You can count the number of lines that have been printed by counting the number of `'\n'` characters that have been displayed. This doesn't always work — why?

Exercise 28 - Passing Parameters to `main()`

The MS-DOS utility echo prints its parameters. For example, typing

```
echo hello world
```

will produce the output:

```
hello world
```

Write a program which will perform this task.

Exercise 29 - Using `unlink()` to Delete Files

The following program will delete a file from your directory (use with care!).

```
int main(int argc, char *argv[])
   {
   unlink(argv[1]);
   return 0;
   }
```

Modify the program to check that a program parameter has been supplied, to print the name of the file it is just about to delete, and to ask the user for confirmation before the file is deleted.

Modify the program so that it will accept a list of file names as parameters to the program and then will step through each file in turn deleting the file if the user confirms the delete.

Exercise 30 - Structures and Arrays of Structures

Write a program that stores the names, two line addresses and ages of a group of people. Each person should be stored in a structure. Use an array of such structures to hold the data for the whole group. Write functions for reading a person's details from the standard input stream and for printing a person.

Exercise 31 - Matrix Multiplication, `typedef`

Matrix multiplication is defined as

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} * \begin{vmatrix} e & f \\ g & h \end{vmatrix} = \begin{vmatrix} a*e+b*g & a*f+b*h \\ c*e+d*g & c*f+d*h \end{vmatrix}$$

Define a new type called `matrix` with

```
typedef  float  matrix[2][2];
```

Write a function which will take two matrices as parameters and return the product through a third parameter.

Exercise 32 - Preprocessor Macros

Define a macro `swap(t, x, y)` that interchanges two variables of the type specified, e.g.

```
int   i = 10, j = 64;
float x = 1.23, y = 2.34

swap(int,i,j);
fswap(float,x,y);
```

Exercise 33 - Pointers to Functions

Write a function which will tabulate values of any given function from a given starting value up to (and including) a given final value. The size of the each step from the starting value to the final value should also be a parameter of the function.

For example,

```
#define PI   3.14159

tab_func(sin, 0.0, PI, PI/8.0);
```

would print

```
        x         f(x)

   0.0000       0.0000
   0.3927       0.3827
   0.7854       0.7071
   1.1781       0.9239
   1.5708       1.0000
   1.9635       0.9239
   2.3562       0.7071
   2.7489       0.3827
   3.1416       0.0000
```

19 Operator Precedence Table

Operator	Associativity
`() [] -> .`	left
`! ~ ++ -- - (type) * & sizeof`	right
`* / %`	left
`+ -`	left
`<< >>`	left
`< <= > >=`	left
`== !=`	left
`&`	left
`^`	left
`|`	left
`&&`	left
`||`	left
`?:`	right
assignments	right
`,`	left