Programming in C

Dr. Neill Campbell

Room 3.14 MVB

Neill.Campbell@bristol.ac.uk

http://www.cs.bris.ac.uk/

About the Course

The course notes were originally based on:
C By Dissection (3rd edition)
Al Kelley and Ira Pohl
because I liked arrays being taught later.

Resources

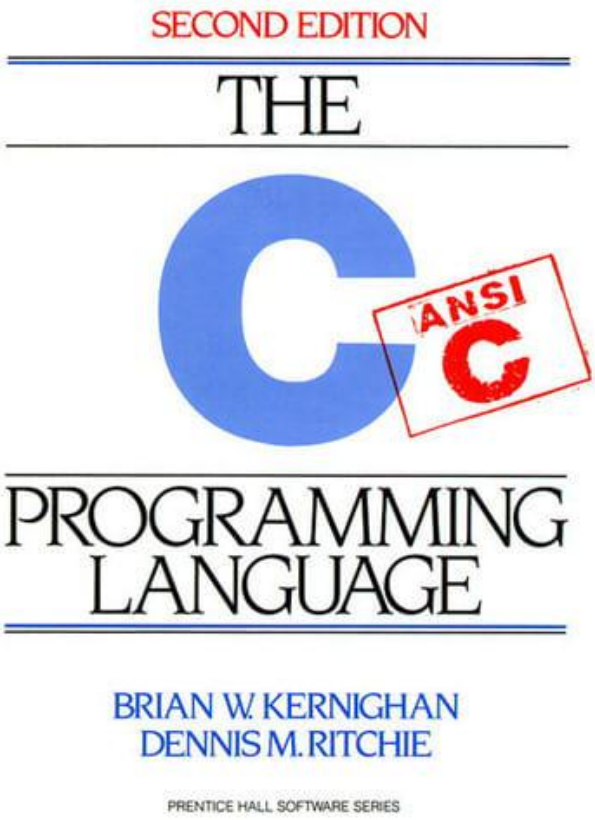
- Free : `https://en.wikibooks.org/wiki/C_Programming`

- A list of more : `https://www.linuxlinks.com/excellent-free-books-learn-c/`

- Whatever you use, make sure it's **ANSI C** that's being taught, not something else e.g. C11 or C++.

K&R

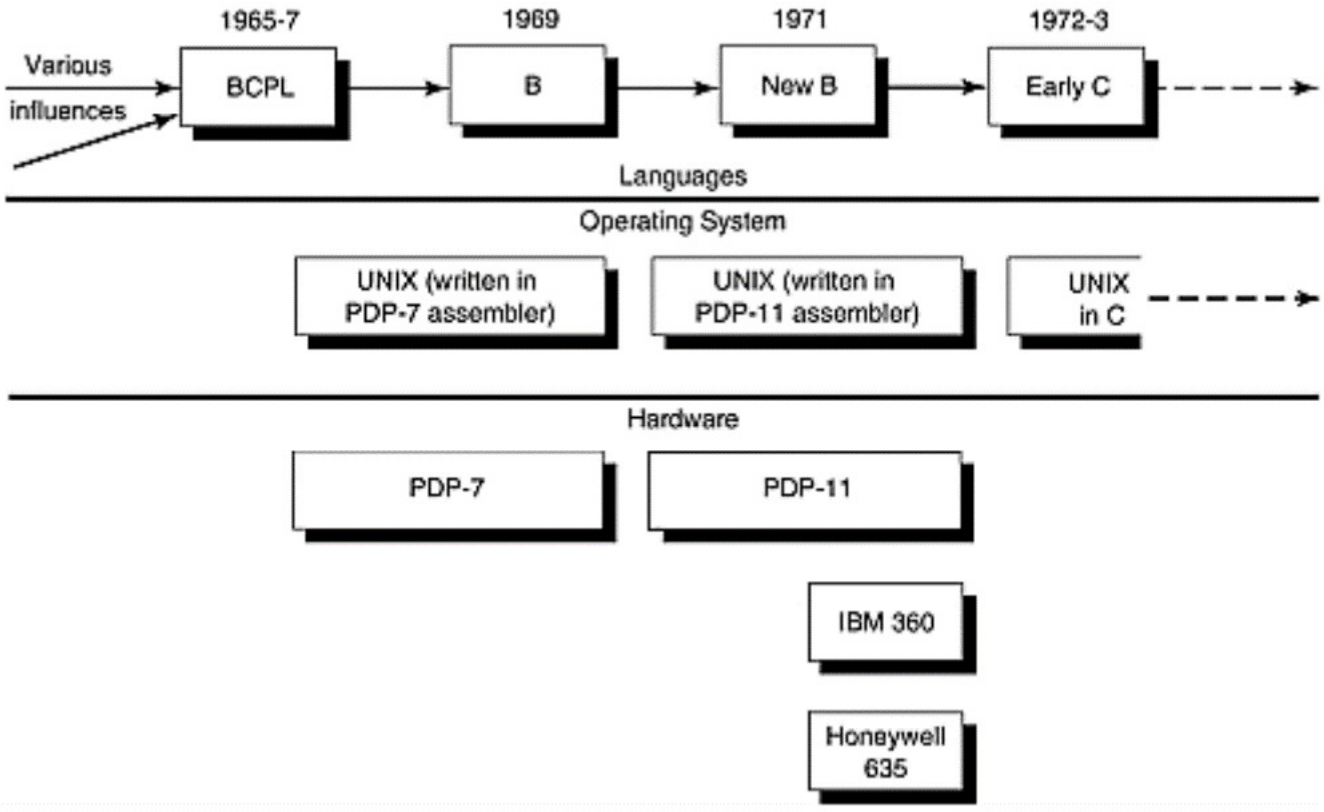
If you fall in love with C and know you're going to use it for the rest of your life, the reference 'bible' is K&R 2nd edition. It's not a textbook for those new to programming, though.



Computer Science Ethos

- Talk to your friends, ask for help, work together.
- Never pass off another persons work as your own.
- Do not pass work to others - either on paper or electronically - even after the submission deadline.
- If someone takes your code and submits it, we need to investigate where it originated - all students involved will be part of this.
- Don’t place your code on publicly accessible sites e.g. github - other students may have extensions etc.

History of C



From *Deep C Secrets* by Peter Van Der Linden

History of C

- BCPL - Martin Richards
- B - Ken Thomson 1970
- Both of above are *typeless*.
- C - Dennis Ritchie 1972 designed for (& implemented on) a UNIX system.
- K&R C (Kernighan and Ritchie) 1978
- ANSI C (COMSM1201)
- C++ - Object Oriented Programming (OOP)
- Java (Subset of C++, WWW enabled).

Why C ?

- Low-level (c.f. Java)
- Doesn’t hide nitty-gritty
- Fast ?
- Wide acceptability
- Large parts common to Java

Programming and Software Engineering

- Was traditionally Lectured 2(or 3) hours a week for weeks 1-12

- With COVID-19 I'll post the equivalent online, broken into manageable chunks

- Programming (C), data structures, algorithms - searching, sorting, string processing, trees etc.

Assessment

- Weekly (unmarked) exercises that, if completed, should ensure you are able to pass the unit.

- Approximately three/four assignments and one lab test.

- One major project due in early TB2 (35%).

- Hard to gauge timings, so don’t make any plans in advance - I’ll change it if we’re going too fast.

Help with Computers

- Any problems with the computers e.g. installing the correct S/W, accessing lab machines: `http://www.bris.ac.uk/it-services/`.

- They are also the people to see about passwords etc.

- This page also links to the rather useful Laptop & Mobile Clinic.

Help with the Unit

- Further information is available via the Blackboard site.

- Help will mainly be via myself giving ‘live’ Q&A session, the associated MS Teams group and the corresponding Forum.

- You will often work in a peer group (approx 15 people).

- There will be a group of Teaching Assistants to help each of these groups.

- TAs are not allowed to write pieces of code for you, nor undertake detailed bug-fixing of your program.

A First Program

```
/* The traditional first program
   in honour of Dennis Ritchie
   who invented C at Bell Labs
   in 1972 */

#include <stdio.h>

int main(void)
{
   printf("Hello, world!\n");
   return 0;
}
```

Dissecting the 1st Program (1)

- Comments are bracketed by the /* and */ pair.

- `#include <stdio.h>`

  Lines that begin with a **#** are called preprocessing directives.

- `int main(void)`

  Every program has a function called

  `main()`

- Statements are grouped using braces,

  `{ ... }`

Dissecting the 1st Program (2)

- `printf()` One of the pre-defined library functions being called (invoked) using a single argument the string:

  `"Hello, world!\n"`

- The `\n` means print the single character *newline*.

- Notice all declarations and statements are terminated with a semi-colon.

- `return(0)` Instruct the Operating System that the function `main()` has completed successfully.

The Second Program

```
#include <stdio.h>

int main(void)
{
   int   inches, feet, fathoms;

   fathoms = 7;
   feet = 6 * fathoms;
   inches = 12 * feet;

   printf("Wreck of the Hesperus:\n");
   printf("Its depth at sea in");
   printf(" different units:\n");
   printf("   %d fathoms\n", fathoms);
   printf("   %d feet\n", feet);
   printf("   %d inches\n", inches);
   return 0;
}
```

Dissecting the 2nd Program

- `#include <stdio.h>` Always required when using I/O.
- `int inches, feet, fathoms;` *Declaration*
- `fathoms = 7;` *Assignment*
- `printf()` has 2 Arguments. The *control string* contains a `%d` to indicate an integer is to be printed.

```
preprocessing directives

int main(void)
{
   declarations

   statements
}
```

Arithmetic Operators

- `+ , - , / , *, %`

- Addition, Subtraction, Division, Multiplication, Modulus.

- Integer arithmetic discards remainer i.e. `1/2` is 0 , `7/2` is 3.

- Modulus (Remainder) Arithmetic. `7%4` is 3, `12%6` is 0.

- Only available for integer arithmetic.

The Character Type

- The keyword `char` stands for character.

- Used with single quotes i.e. `'A'`, `'+'`.

```
#include <stdio.h>

int main(void)
{
   char   c;

   c = 'A';
   printf("%c\n", c);
   return 0;
}
```

- The letter A is printed.

- Note the `%c` conversion format.

Floating Types

- In ANSI C there are three floating types:

  1. `float`
  2. `double`
  3. `long double`

- The *Working Type* is doubles.

```
#include <stdio.h>

int main(void)
{
   float   x, y;

   x = 1.0;
   y = 2.0;
   printf("Sum of x & y is %f.\n", x + y);
   return 0;
}
```

The Preprocessor

- A `#` in the first column signifies a preprocessor statement.

- `#include <file.h>` Exchange this line for the entire contents of `file.h`, which is to be found in a standard place.

- `#define ROUGH_PI (22.0/7.0)`
  Replace all occurrences of `ROUGH_PI` with `(22.0/7.0)`.

- Include files generally contain other `#define`'s and `#include`'s.

Using printf()

- `printf( fmt-str, arg1, arg2, ...);`

`%c`	Characters
`%d`	Integers
`%e`	Floats/Doubles (Engineering Notation)
`%f`	Floats/Doubles
`%s`	Strings

- Fixed-width fields: `printf("F:%7f\n", f);`
  `F: 3.0001`

- Fixed Precision: `printf("F:%.2f\n", f);`
  `F:3.00`

Using scanf()

- Similar to `printf()` but deals with *input* rather than *output*.

- `scanf(fmt-str, &arg1, &arg2, ...);`

- Note that the *address* of the argument is required.

`%c`	Characters
`%d`	Integers
`%f`	Floats
`%lf`	Doubles
`%s`	Strings

- Note doubles handled differently than floats.

The `while` Loop

```
while (test is true) {
    statement 1;
        ...
    statement n;
}
```

```
/* Sums are computed. */
#include <stdio.h>

int main(void)
{
   int     cnt = 0;
   float   sum = 0.0, x;

   printf("Input some numbers:  ");
   while (scanf("%f", &x) == 1) {
      cnt = cnt + 1;
      sum = sum + x;
   }
   printf("\n%s%5d\n%s%12f\n\n",
      "Count:", cnt,
      "  Sum:", sum);
   return 0;
}
```

Common Mistakes

Missing ”

```
printf("%c\n, ch);
```

Missing ;

```
a = a + 1
```

Missing Address in `scanf()`

```
scanf("%d", a);
```

Chapter Two
Operators

Grammar

- C has a grammar/syntax like every other language.
- It has *Keywords*, *Identifiers*, *Constants*, *String Constants*, *Operators* and *Punctuators*.
- Valid Identifiers :
  `k`, `_id`, `iamanidentifier2`, `so_am_i`.
- **Invalid** Identifiers :
  `not#me`, `101_south`, `-plus`.

- Constants :
  `17` (decimal), `017` (octal), `0x17` (hexadecimal).

- String Constant enclosed in double-quotes:
  `"I am a string"`

Operators

- All operators have rules of both *precedence* and *associativity*.

- `1 + 2 * 3` is the same as `1 + (2 * 3)` because `*` has a higher precedence than `+`.

- The associativity of `+` is left-to-right, thus
  `1 + 2 + 3` is equivalent to `(1 + 2) + 3`.

- Increment and decrement operators:
  `i++;` is equivalent to `i = i + 1;`

- May also be prefixed `--i;`

```
int a, b, c = 0;
a = ++c;
b = c++;
printf("%d %d %d\n", a, b, ++c);
```

Assignment

- The = operator has a low precedence and a right-to-left associativity.

- `a = b = c = 0;` is valid and equivalent to:
  `a = (b = (c = 0));`

- `i = i + 3;` is the same as `i += 3;`

- Many other operators are possible e.g.

  `-=, *=, /=.`

The Power of 2 Program

```
/* Some powers of 2 are printed. */

#include <stdio.h>

int main(void)
{
   int   i = 0, power = 1;

   while (++i <= 10)
      printf("%5d", power *= 2);
   printf("\n");
   return 0;
}
```

The output is:

```
    2    4    8   16   32   64  128  256  512 1024
```

Random Numbers

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

int main(void)
{
   int   i, n;

   printf("\n%s\n%s",
    "Randomly distributed integers"
    "are printed." ,
    "How many do you want to see?  ");
   scanf("%d", &n);
   for (i = 0; i < n; ++i) {
      if (i % 4 == 0)
         printf("\n");
      printf("%12d", rand());
   }
   printf("\n");
   return 0;
}
```

The Standard Library

The output is:

```
Some randomly distributed integers will be printed.
How many do you want to see?  11

    16838     5758    10113    17515    31051     5627
    23010     7419    16212     4086     2749
```

- Definitions required for the proper use of many functions such as `rand()` are found in `stdlib.h`.

- Do not mistake these header files for the libraries themselves !

Chapter Three
Flow Control

Relations

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	**equal to**
!=	not equal to
!	not
&&	logical and
||	logical or

True and False

- Any relation is either *true* or *false*.
- Any non-zero value is *true*.
- `(a < b)` returns the value *0* or *1*.
- `(i == 5)` is a **test** not an **assignment**.
- `(!a)` is either *true (1)* or *false (0)*.
- `(a && b)` is *true* if both `a` and `b` are *true*.

Short-Circuit Evaluation

```
if(x >= 0.0 && sqrt(x) < 10.0){

..... /* Do Something */

}
```

The `sqrt()` statement is never reached if the first test is *false*, since in a logical AND, once any expression is *false* the whole must be *false*.

The `if()` Statement

```
if (expr)
   statement
```

If more than one statement is required:

```
if (expr) {
   statement-1
        .
        .
        .
   statement-n
}
```

Adding an `else` statement:

```
if (expr) {
   statement-1
        .
        .
        .
   statement-n
}
else{
   statement-a
        .
        .
        .
   statement-e
}
```

A practical example:

```
#include <stdio.h>

int main(void)
{
   int   x, y, z, min;

   printf("Input three integers:  ");
   scanf("%d%d%d", &x, &y, &z);
   if (x < y)
      min = x;
   else
      min = y;
   if (z < min)
      min = z;
   printf("The minimum value is " \
          "%d\n", min);
   return 0;
}
```

The `while()` Statement

```
while(expr)
    statement
```

This, as with the `for` loop, may execute compound statements:

```
while(expr){
    statement-1
         .
         .
         .
    statement-n
}
```

The `for()` Loop

This is one of the more complex and heavily used means for controlling execution flow.

```
for( init ; test; loop){
   statement-1
        .
        .
        .
   statement-n
}
```

and may be thought of as:

```
init;
while(test){
    statement-1
          .
          .
          .
    statement-n
    loop;
}
```

In the `for()` loop, note:

- Semi-colons separate the three parts.
- Any (or all) of the three parts could be empty.
- If the test part is empty, it evaluates to *true*. `for(;;){ a+=1; }` is an infinite loop.

```
/* Find triples of integers that
   add up to N.                    */

#include <stdio.h>

#define   N   7

int main(void)
{
   int   cnt = 0, i, j, k;

   for (i = 0; i <= N; i++)
      for (j = 0; j <= N; j++)
       for (k = 0; k <= N; k++)
          if (i + j + k == N) {
             ++cnt;
             printf("%3d%3d%3d\n",
                    i, j, k);
          }
   printf("\nCount: %d\n", cnt);
   return 0;
}
```

The result is:

```
0   0   7
0   1   6
0   2   5
0   3   4
0   4   3
0   5   2
0   6   1
0   7   0
1   0   6
1   1   5
1   2   4
1   3   3
1   4   2
1   5   1
1   6   0
2   0   5
2   1   4
2   2   3
2   3   2
2   4   1
2   5   0
3   0   4
3   1   3
3   2   2
3   3   1
3   4   0
4   0   3
4   1   2
4   2   1
4   3   0
5   0   2
5   1   1
5   2   0
6   0   1
6   1   0
7   0   0
```

The Comma Operator

This has the lowest predence of all the operators in C and associates left to right.

```
a = 0 , b = 1;
```

Hence, the `for` loop may become quite complex :

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

An equivalent, but more difficult to read expression :

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

- Notice the loop has an empty body, hence the semicolon.

The `do-while()` Statement

```
do {
   statement-1
        .
        .
        .
   statement-n
} while ( test );
```

Unlike the `while()` loop, the `do-while()` will always be executed at least once.

The `switch()` Statement

```
switch (val) {
   case 1 :
      a++;
      break;
   case 2 :
   case 3 :
      b++;
      break;
   default :
      c++;
}
```

- The `val` must be an integer.
- The `break` statement causes execution to jump out of the loop. No `break` statement causes execution to 'fall through' to the next line.
- The `default` label is a catch-all.

The Conditional Operator

```
expr1 ? expr2 : expr3
```

If `expr1` is *true* then `expr2` is executed, else `expr3` is evaluated, i.e.:

```
x = ((y < z) ? y : z);
```

Chapter Four
Functions

```
#include <stdio.h>

int   min(int a, int b);

int main(void)
{
   int   j, k, m;

   printf("Input two integers:  ");
   scanf("%d%d", &j, &k);
   m = min(j, k);
   printf("\nOf the two values " \
          " %d and %d, " \
          "the minimum is %d.\n\n",
          j, k, m);
   return 0;
}
```

```
int min(int a, int b)
{
   if (a < b)
      return a;
   else
      return b;
}
```

- Execution begins, as normal, in the `main()` function.

- The function *prototype is shown* at the top of the file. This allows the compiler to check the code more thoroughly.

- The function `min()` returns an `int` and takes two `int`'s as arguments.

- The function is defined between two braces.

- The `return` statement is used return a value to the calling statement.

- A function which has no return value, is declared `void` and is equivalent to a procedure.

Assert

The assert macro is defined in the header file `assert.h`. This is used to ensure the value of an expression is as we expect it to be.

```
#include <assert.h>

double f(int a, int b)
{

   double x;

   assert(a > 0);

   /* precondition */
   assert(b >= 7 && b <= 11);

      .
      .
      .

   /* postcondition */
   assert(x >= 1.0);

   return(x);
}
```

- If an assertion fails, an error is printed and the program is aborted.

- By `#define`'ing `NDEBUG` all assertions are ignored, allowing them to be used during development and switched off later.

Program Layout

It is common for the `main()` function to come first in a program :

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

list of function prototypes

int main(void)
{
   . . . . .
}

int f1(int a, int b)
{
   . . . . .
}

int f2(int a, int b)
{
   . . . . .
}
```

However, it is possible to avoid the need for function prototypes by defining a function before it is used :

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

int f1(int a, int b)
{
   . . . . .
}

int f2(int a, int b)
{
   . . . . .
}

int main(void)
{

   . . . . .

}
```

Call by Value

In the following example, a function is passed an integer using call by value:

```
#include <stdio.h>

void fnc1(int a);

int main(void)
{

    int x = 1;

    fnc1(x);
    printf("%d\n", x);

}
```

```
void fnc1(int a)
{

   a = a + 1;

}
```

The function does not change the value of `x` in `main()`, since `a` in the function is effectively only a **copy** of the variable.

Multiply

Write a simple function `int mul(int a, int b)` which multiples two integers together without the use of the multiply symbol in C (i.e. the `*`).

Use `assert()` calls in `main()` test it thoroughly.

Recursion

A repeated computation computation is normally achieved via *iteration*, e.g. using `for()`:

```
#include <stdio.h>

int fact(int a);

int main(void)
{

   int a, f;

   printf("Input a number :\n");
   scanf("%d", &a);
   f = fact(a);
   printf("%d! is %d\n", a, f);

   return(0);

}
```

```
int fact(int a)
{

   int i;
   int tot = 1;

   for(i=1; i<=a; i++){
      tot *= i;
   }
   return tot;
}
```

We could also achieve this via *recursion* :

```c
#include <stdio.h>

int fact(int a);

int main(void)
{

   int a, f;

   printf("Input a number :\n");
   scanf("%d", &a);
   f = fact(a);
   printf("%d! is %d\n", a, f);

   return(0);
}

int fact(int a)
{
   if(a > 0)
      return (a*fact(a - 1));
   else
      return 1;

}
```

Chapter Five
Characters

Storage of Characters

- Characters are stored in the machine as one byte i.e. one of **256** possible values.

- These may be thought of a characters, or very small integers.

- Only a subset of these 256 values are required for the printable characters, space, newline etc.

- Declaration:

```
char c;

c = 'A';
```

or :

```
char c1 = 'A', c2 = '*', c3 = ';' ;
```

The particular integer used to represent a character is dependent on the encoding used. The most common of these, used on most UNIX and PC platforms, is ASCII.

lowercase	'a'	'b'	'c'	. . .	'z'
ASCII value	97	98	99	. . .	112
uppercase	'A'	'B'	'C'	. . .	'Z'
ASCII value	65	66	67	. . .	90
digit	'0'	'1'	'2'	. . .	'9'
ASCII value	48	49	50	. . .	57
other	'&'	'*'	'+'	. . .	
ASCII value	38	42	43	. . .	

- When using `printf()` and `scanf()` the formats `%c` and `%d` do very different things :

```
char c = 'a'
printf("%c\n", c); /* prints : a  */
printf("%d\n", c); /* prints : 97 */
```

- Hard-to-print characters have an escape sequence i.e. to print a newline, the 2 character escape `'\n'` is used.

Using `getchar()` and `putchar()`

```
/* Outputs characters twice */
#include <stdio.h>

int main(void)
{

   char c;
   while (1) {
      c = getchar();
      putchar(c);
      putchar(c);
   }
   return 0;
}
```

This has the unfortunate problem of never terminating, a CTRL-C would be required.

```
#include <stdio.h>

int main(void)
{
   int   c;

   while ((c = getchar()) != EOF) {
      putchar(c);
      putchar(c);
   }
   return 0;
}
```

The end-of-file constant is defined in stdio.h. Although system dependent, -1 is often used. On the UNIX system this is generated when the end of a file being piped is reached, or when CTRL-D is pressed.

Capitalising

```
/* Outputs characters twice */

#include <stdio.h>

#define CAPS ('A' - 'a')

int main(void)
{
   int   c;

   while ((c = getchar()) != EOF)
      if (c >= 'a' && c <= 'z')
          putchar(c + CAPS);
      else
         putchar(c);
   return 0;
}
```

This is more easily achieved by using some of the definitions found in `ctype.h`.

Macro	`true` returned if:
isalnum(int c)	Letter or digit
isalpha(int c)	Letter
iscntrl(int c)	Control character
isdigit(int c)	Digit
isgraph(int c)	Printable (not space)
islower(int c)	Lowercase
isprint(int c)	Printable
ispunct(int c)	Punctuation
isspace(int c)	White Space
isupper(int c)	Uppercase
isxdigit(int c)	Hexadecimal
isascii(int c)	ASCII code

Some useful functions are :

Function/Macro	Returns:
int tolower(int c)	Lowercase c
int toupper(int c)	Uppercase c
int toascii(int c)	ASCII code for c

A better version of capitalise is now :

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

int main(void)
{
   int   c;

   while ((c = getchar()) != EOF)
      if (islower(c))
         putchar(toupper(c));
      else
         putchar(c);
   return 0;
}
```

Chapter Six
Data Types

Fundamental Data types

[unsigned | signed**]** **[**long | short**]** **[**int | float | double**]**

char	signed char	unsigned char
signed short int	signed int	signed long int
unsigned short int	unsigned int	unsigned long int
float	double	long double

The use of `int` implies `signed int` without the need to state it. Likewise `unsigned short` means `unsigned short int`.

Binary Storage of Numbers

In an unsigned char :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	0	0	1	1	0	0

The above represents : $1 * 64 + 1 * 8 + 1 * 4 = 76$.

Type	Num Bytes	Minimum	Maximum
signed char	1	-128	127
unsigned char	1	0	255
signed short	2 (Often)	-32768	32767
unsigned short	2 (Often)	0	65535
signed int	4 (Often)	-2147483648	2147483647
unsigned int	4 (Often)	0	4294967295
signed float	4 (Often)	10^{-38}	10^{+38}

- Not all floats are representable so are only approximated.

- Floating operations need not be exact.

To find the exact size in bytes of a type on a particular machine, use `sizeof(type)`. On a Dell Windows 10 laptop running WSL:

```
#include <stdio.h>

int main(void)
{
   printf("\n");
   printf("char\t\t:%3ld\n", sizeof(char));
   printf("short\t\t:%3ld\n", sizeof(short));
   printf("int\t\t:%3ld\n", sizeof(int));
   printf("long\t\t:%3ld\n", sizeof(long));
   printf("unsigned\t:%3ld\n", sizeof(unsigned));
   printf("float\t\t:%3ld\n", sizeof(float));
   printf("dbl\t\t:%3ld\n", sizeof(double));
   printf("long dbl\t:%3ld\n", sizeof(long double));
   printf("\n");

   return 0;

}
```

results in :

```
char            :  1
short           :  2
int             :  4
long            :  8
unsigned        :  4
float           :  4
dbl             :  8
long dbl        : 16
```

Mathematical Functions

There are no mathematical functions built into the C language. However, there are many functions in the maths library which may linked in using the **-lm** option with the compiler.

Functions include :

```
sqrt()  pow()  exp()  log()
 sin()  cos()  tan()
```

Most take `double`'s as arguments and
return `double`'s. A demonstration of one of the functions is shown :

```
/* Precision Demo */
#include <stdio.h>
#include <math.h>

int main(void)
{

   int i;
   double x, y;

   printf("Enter a Number : ");
   scanf("%lf", &x);
   y = x;
   for(i=0; i< 100000; i++){
      y = pow(y, 80.0);
      y = pow(y, 1.0/80.0);
   }
   printf("%.10f %.10f\n", x, y);
   return 0;

}

Enter a Number :  1234.5678
1234.5678000000 1234.5678000455
```

Casting

An explicit type conversion is called a *cast*. For instance if we need to find the cube of an integer `i`:

```
k = (int) pow( (double) i , 3.0);
```

Chapter Seven
Enumeration & Typedefs

Enumerated Types

```
enum day { sun, mon, tue, wed, thu, fri, sat};
```

- This creates a user-defined **type** `enum day`.
- The enumerators are constants of type `int`.
- By default the first (`sun`) has the value `0`, the second has the value `1` and so on.
- An example of their use:

```
enum day d1;
 . . .
d1 = fri;
```

- The default numbering may be changed as well:

```
enum fruit{apple=7, pear, orange=3, lemon};
```

- Use enumerated types as constants to aid readability - they are self-documenting.

- Declare them in a header (`.h`) file.

- Note that the type is `enum day`; the keyword `enum` is not enough.

Typedefs

- Sometimes it is useful to associate a particular name with a certain type, e.g.:

```
typedef int colour;
```

- Now the type `colour` is synonymous with the type `int`.

- Makes code self-documenting.

- Helps to control complexity when programmers are building complicated or lengthy user-defined types (See **structures** Chapter 12).

Combining `typedef`s and `enum`s

- Often `typedef`'s are used in conjunction with enumerated types:

```
enum day {sun,mon,tue,wed,thu,fri,sat};

typedef    enum day     day;

day find_next_day(day d)
{

   day next_day;

   switch(d){
      case sun:
         next_day = mon;
         break;
      case mon:
         next_day = tue;
         break;
      . . . .
      case sat:
         next_day = sun;
         break;
      default:
         printf("I wasn't expecting " \
                "that !\n");
   }
   return next_day;
}
```

Style

```
enum veg {beet, carrot, pea};
typedef enum veg veg;
veg v1, v2;
v1 = carrot;
```

- We can combine the two operations into one:

```
typedef enum veg {beet,carrot,pea} veg;
veg v1, v2;
v1 = carrot;
```

- Assigning:

```
v1 = 10;
```

is very poor programming style !

Fever

Rewrite/complete this code using `typedefs` and `enum`s to create self-documenting code in any manner you wish.

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

int fvr(double t, int s);

int main(void)
{
   assert(fvr(37.5, 0)==1);
   assert(fvr(36.5, 0)==0);
   assert(fvr(96.5, 1)==0);
   assert(fvr(99.5, 1)==1);

   return 0;
}

/* Argument 1 is temperature
   Argument 2 is scale (0=>Celsius, 1=>Farenheit)
*/
int fvr(double t, int s)
{
}
```

Chapter Eight
Pointers

Call-by-Value

```
#include <stdio.h>

void change(int i);

int main(void)
{
   int v = 1;
   change(v);
   printf("%d\n", v);
}

void change(int v)
{
   v = 2;
}
```

In the program, the function cannot change the value of `v` as defined in `main()` since a **copy** is made of it.

To allow a function to modify the value of a variable passed to it we need a mechanism known as **call-by-reference**, which uses the **address** of variables (pointers).

Pointers

We have already seen addresses used with `scanf()`. The function call:

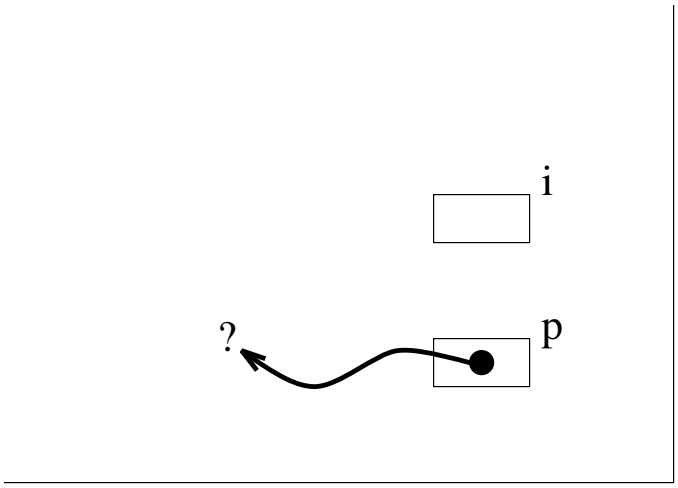
```
scanf("%d", &v);
```

causes the appropriate value to be stored at a particular address in memory. If `v` is a variable, then `&v` is its address, or location, in memory.

Declaring Pointers

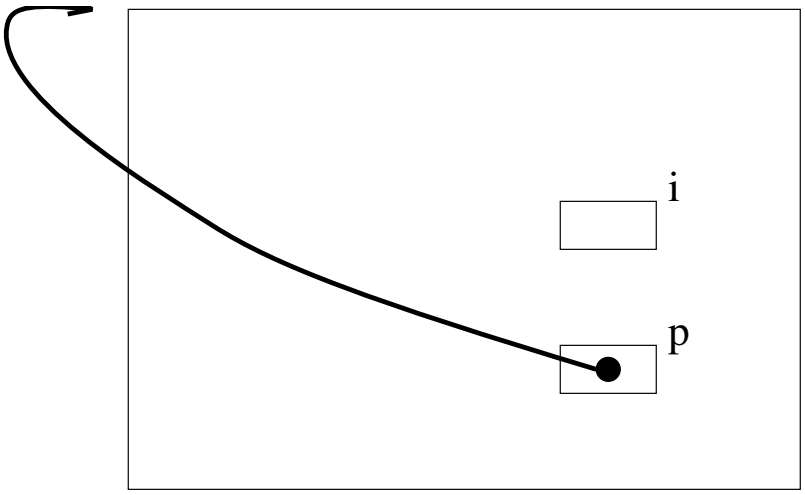
```
int i, *p;
```

Here `i` is an `int` and `p` is of type *pointer to int*. Pointers have a legal range which includes the special address `0` and a set of positive integers which are the machine addresses of a particular system.

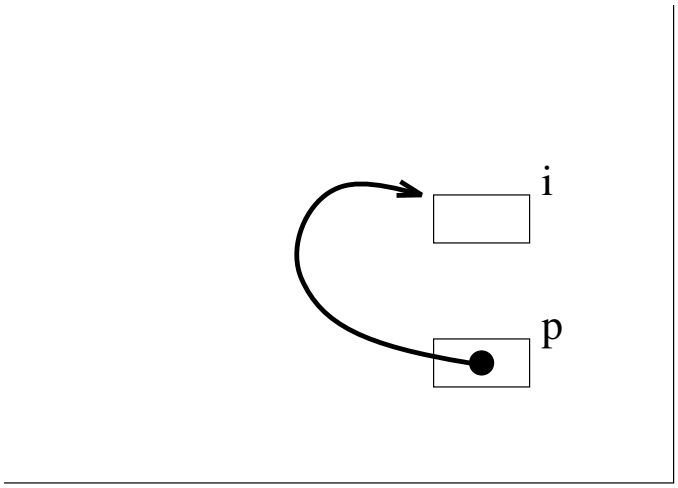


Assigment

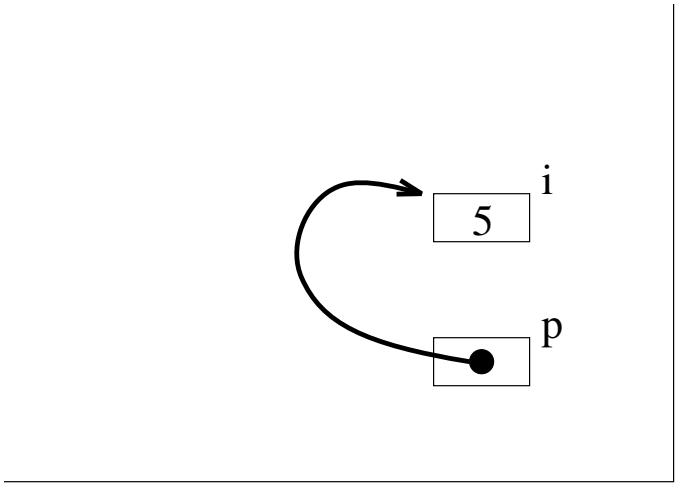
```
p = NULL;
```



```
p = &i;
```




```
i = 5;
```



Dereferencing Pointers

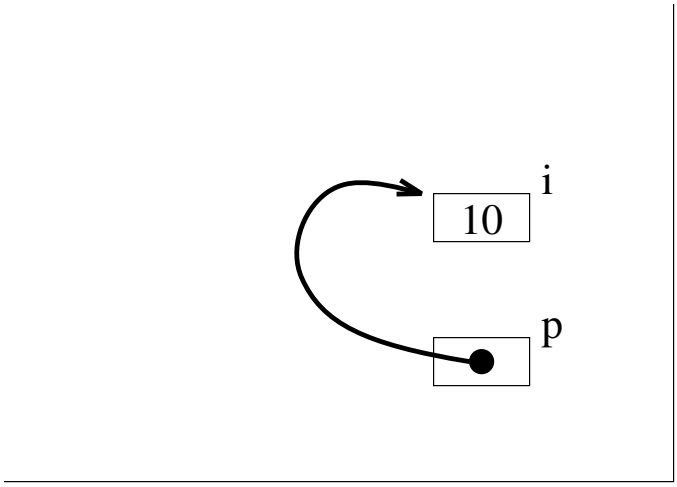
```
printf("%d\n", *p);/* Dereference p */
```

This prints `5`.

```
i = 17;
printf("%d\n", *p);
```

This prints `17`.

```
*p = 10;
```




```
printf("%d\n", i);
```

This prints `10`.

Back to `scanf()`

Using:

```
scanf("%d", &i);
```

is now the same as:

```
scanf("%d", p);
```

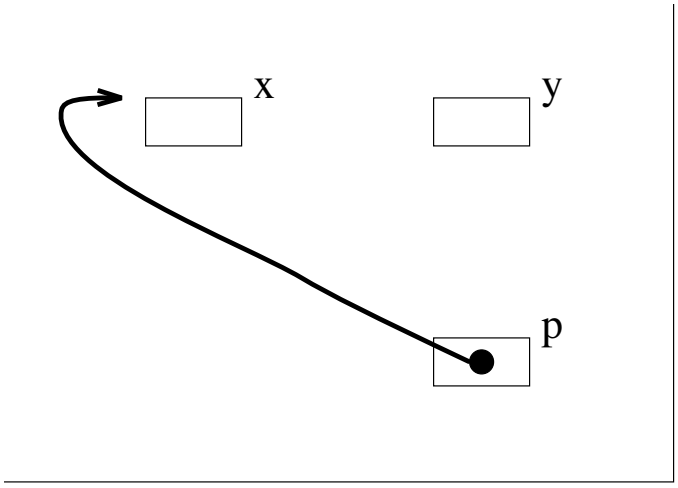
provided that `p = &i`.

In many ways the dereference operator `*` is the inverse of the address operator `&`.

Equivalence

```
float x, y, *p;

p = &x;
y = *p;
```



What is this equivalent to ?

Pointers may be assigned when both sides have the same type:

```
int *p;
p = 10;              /* Illegal */
p = (int *) 10;      /*  Legal  */
```

The Swap Function

```
#include <stdio.h>

void swap(int *p, int *q);

int main(void)
{
   int   a = 3, b = 7;

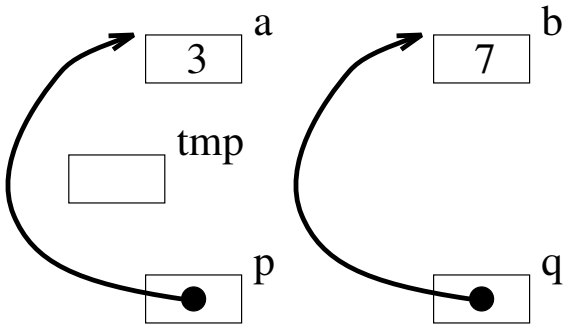
   /* 3 7 printed */
   printf("%d %d\n", a, b);
   swap(&a, &b);
   /* 7 3 printed */
   printf("%d %d\n", a, b);
   return 0;
}

void swap(int *p, int *q)
{
   int   tmp;

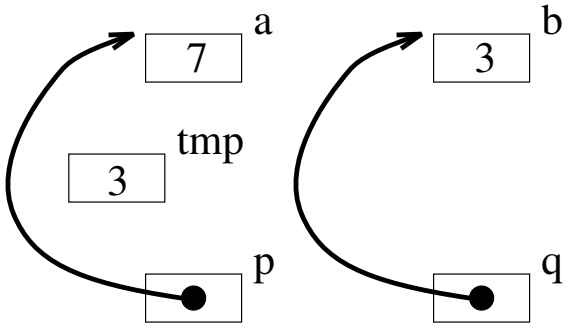
   tmp = *p;
   *p = *q;
   *q = tmp;
}
```

Dissecting `swap()`

When the function is called:




```
tmp = *p;
*p = *q;
*q = tmp;
```



Storage Classes

- **auto**

  ```
  auto int a, b, c;
  auto float f;
  ```

  Because this is the default, it is seldom used.

- **extern**

  Tells the compiler to look for the variable elsewhere, possibly another file.

- **register**

  Informs the compiler to place the variable in a high-speed memory register if possible,
  i.e. if there are enough such registers available & the hardware supports this.

- **static**

  Allows local variables to maintain their values :

```
void f(void)
{

   static int cnt = 0;

   cnt++;
}
```

Type Qualifiers

- **const**

  ```
  const int k = 3;
  ```

  Once `k` is initialised, it cannot be changed in any way.

- **volatile**

  ```
  extern const volatile int real_time_clock;
  ```

  Very rarely used, this variable may be changed by the *hardware* in some unspecified way.

Chapter Nine
Arrays & Pointers

One-Dimensional Arrays

```
#include <stdio.h>

#define   N   5

int main(void)
{
   /* allocate space a[0]...a[4] */
   int   a[N];
   int   i, sum = 0;

   /* fill array */
   for (i = 0; i < N; ++i)
      a[i] = 7 + i * i;
   /* print array */
   for (i = 0; i < N; ++i)
      printf("a[%d] = %d", i, a[i]);
   /* sum elements */
   for (i = 0; i < N; ++i)
      sum += a[i];
   /* print sum */
   printf("\nsum = %d\n", sum);
   return 0;
}
```

- One-Dimensional arrays are declared by a type followed by an identifier with a bracketed constant expression:

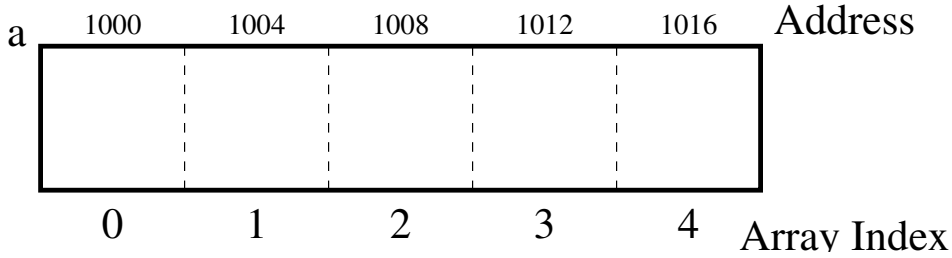
```
float x[10];
int k[ARRAY_SIZE];
```

  The following, however, is not valid:

```
float y[i*2];
```

- Arrays are stored in contiguous memory, e.g.:

```
int a[5];
```



- Arrays are indexed **0** to **n-1**.

Initialisation

By default, arrays are uninitialised. When they are declared, they may be assigned a value:

```
float x[7] = {-1.1,0.2,2.0,4.4,6.5,0.0,7.7};
```

or,

```
float x[7] = {-1.1, 0.2};
```

the elements 2 ... 6 are set to zero.

Also:

```
int a[] = {3, 8, 9, 1};
```

is valid, the compiler assumes the array size to be `4`.

Using Arrays

- Accessing an array out of bounds will not be identified by the compiler. It may cause an error at run-time. One frequent result is that an entirely unrelated variable is altered.

- `a[5] = a[4] + 1;`

- `k[9]++;`

- `n[12+i] = 0;`

```
/* Count lowercase letters */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
   int   c, i, letter[26];

   /*  zero   */
   for (i = 0; i < 26; ++i)
      letter[i] = 0;
   /*  count  */
   while ((c = getchar())!=EOF)
      if (islower(c))
         ++letter[c - 'a'];
   /* results */
   for (i = 0; i < 26; ++i) {
      if (i % 5 == 0)
         printf("\n");
      printf("%c:%4d  ", 'a' + i,
               letter[i]);
   }
   printf("\n\n");
   return 0;
}
```

The results of running the code on the handout of lectures 1 to 7:

```
a: 840  b: 212  c: 436  d: 409  e:1449
f: 313  g: 173  h: 392  i:1112  j:   9
k:  34  l: 470  m: 373  n: 952  o: 666
p: 285  q:  21  r: 743  s: 696  t:1196
u: 398  v: 157  w:  97  x:  56  y: 148
z:  43
```

Arrays are Pointers ?

An array name by itself is simply an address. For instance:

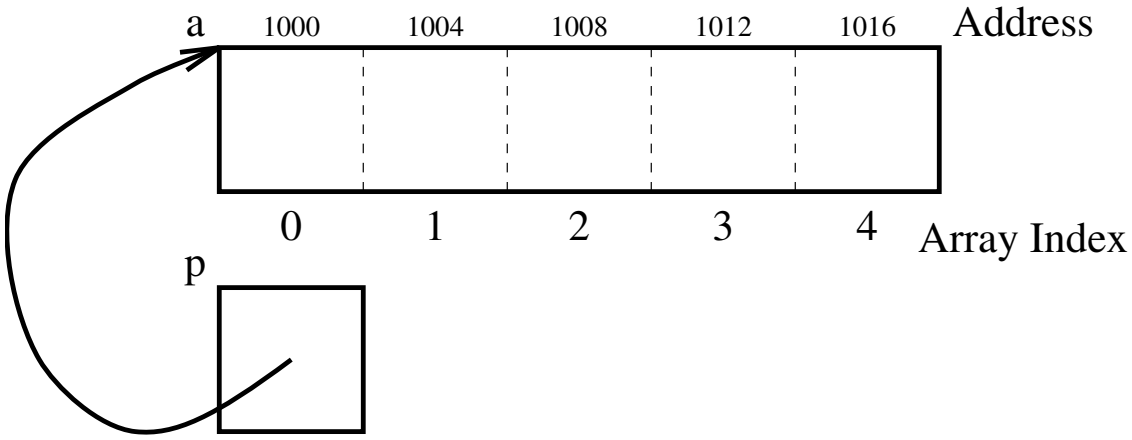
```
int a[5];
int *p;
```

declares an array of `5` elements, and `a` is the address of the start of the array. Assigning:

```
p = a;
```

is completely valid and the same as:

```
p = &a[0];
```



To assign `p` to point to the next element, we could use

```
p = a + 1;
```

or

```
p = &a[1];
```

- Notice that `p = a + 1` advances the pointer **4** bytes and not 1 byte. This is because an integer is 4 bytes long and `p` is a pointer to an int.

- we can use the pointer `p` is exactly the same way as normal, i.e.:

```
*p = 5;
```

Summing an Array - 1

```
#include <stdio.h>

int main(void)
{

   int a[] = {10, 12, 6, 7, 2};
   int i;
   int sum = 0;

   for(i=0; i<5; i++){
      sum += a[i];
   }
   printf("%d\n", sum);

   return 0;

}
```

Summing an Array - 2

```
#include <stdio.h>

int main(void)
{

   int a[] = {10, 12, 6, 7, 2};
   int i;
   int sum = 0;

   for(i=0; i<5; i++){
      sum += *(a + i);
   }
   printf("%d\n", sum);

   return 0;

}
```

Summing an Array - 3

```
#include <stdio.h>

int main(void)
{

   int a[] = {10, 12, 6, 7, 2};
   int i;
   int sum = 0;
   int *p;

   p = a;
   for(i=0; i<5; i++){
      sum += *p;
      p++;
   }
   printf("%d\n", sum);

   return 0;

}
```

Passing Arrays to Functions

When an array is passed to a function, what is actually passed is a pointer to the base address, not a copy of the array itself.

```
#include <stdio.h>

void zero_array(int b[], int s);
void zero_array2(int *b, int s);

int main(void)
{

   int a[5]; /* Uninitialised */

   zero_array(a, 5);
   return 0;

}
```

```
void zero_array(int b[], int s)
{

   int i;
   for(i=0; i<s; i++)
      b[i] = 0;
}

void zero_array2(int *b, int s)
{

   int i;
   for(i=0; i<s; i++)
      *b++ = 0;
}
```

To initialise the middle three elements we could have called:

```
zero_array(&a[1], 3);
```

The Bubblesort

```
#include <stdio.h>

void bubble_sort(int b[], int s);

int main(void)
{

   int i;
   int a[] = {3, 4, 1, 2, 9, 0};

   bubble_sort(a, 6);

   for(i=0; i<6; i++){
      printf("%d ", a[i]);
   }
   printf("\n");

   return 0;

}
```

```
void bubble_sort(int b[], int s)
{

   int i, tmp;
   int changes;

   do{
      changes = 0;
      for(i=0; i<s-1; i++)
         if(b[i] > b[i+1]){
            tmp = b[i];
            b[i] = b[i+1];
            b[i+1] = tmp;
            changes++;
         }
   }while(changes);
}
```

During each pass, the array contains:

```
3 1 2 4 0 9
1 2 3 0 4 9
1 2 0 3 4 9
1 0 2 3 4 9
0 1 2 3 4 9
0 1 2 3 4 9
```

Two-Dimensional Arrays

A 2D array is declared as follows:

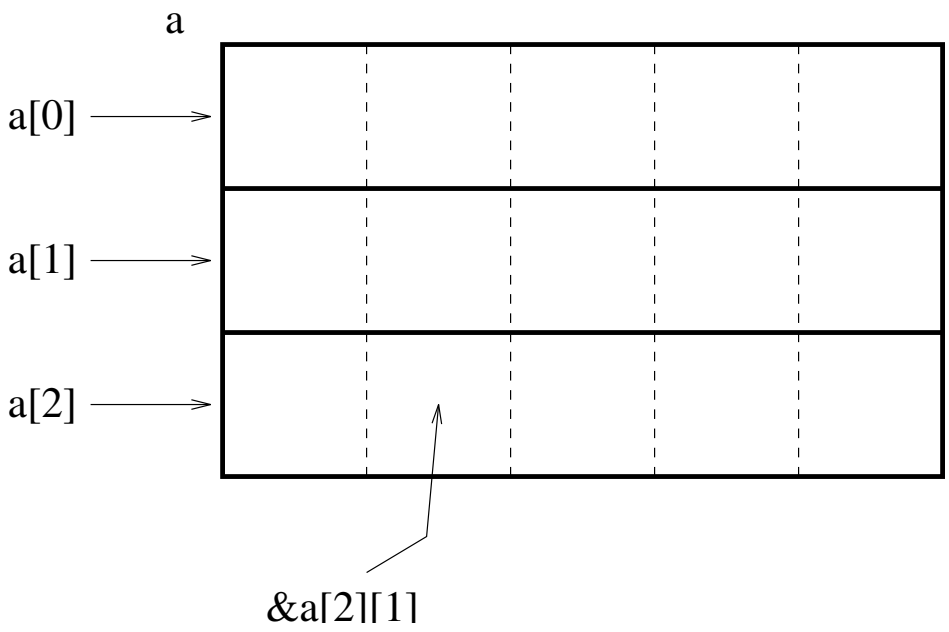
```
#define ROWS 3
#define COLS 5
int a[ROWS][COLS];
```

	col 1	col 2	col 3	col 4	col 5
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

2D array initialisation :

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

Although `a` is stored in a contiguous block of memory, we may think of it as a 2D rectangle of data.



`a[i][j]` is equivalent to:

```
*(a[i] + j)
(*(a + i))[j]
*((*(a + i)) + j)
*(&a[0][0] + 5*i + j)
```

```
#include <stdio.h>

/* number of rows */
#define   M   3
/* number of columns */
#define   N   4

int main(void)
{
   int   a[M][N], i, j, sum = 0;

   putchar('\n');
   /* fill array */
   for (i = 0; i < M; ++i)
      for (j = 0; j < N; ++j)
         a[i][j] = i + j;
   /* array values */
   for (i = 0; i < M; ++i){
      for (j = 0; j < N; ++j)
         printf("a[%d][%d] = %d ",
                  i, j, a[i][j]);
```

```
        printf("\n");
    }
    /* sum the array */
    for (i = 0; i < M; ++i)
        for (j = 0; j < N; ++j)
            sum += a[i][j];
    printf("\nsum = %d\n\n", sum);
    return 0;
}
```

Dynamic Memory

In some cases the size of an array may not be known. The size may be decided by the user, or by a computation. There are two functions which provide the ability to dynamically allocate memory:

```
void *calloc(num_els, el_size);
void *malloc(num_bytes);
```

Both functions return pointers. `calloc()` initialises its memory to zero, whereas `malloc()` does not. If `NULL` is returned, then the allocation has failed.

The file `stdlib.h` contains definitions and prototypes required to use the functions.

The function:
`void free(void *ptr);`
frees the memory once it is no longer required.

```
/* To allocate 256 bytes of memory */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{

   int *p;

   p = (int *)malloc( 256 );

   if(p == NULL){
      printf("Allocation Failed...\n");
      exit(EXIT_FAILURE);
   }

   return(0);

}
```

Chapter Ten
Strings & Pointers

Strings

- Strings are 1D arrays of characters.

- Any character in a string may be accessed as an array element, or by dereferencing a pointer.

- The important difference between strings and ordinary arrays is the **end-of-string sentinel** `'\0'` or null character.

- The string ”abc” has a *length* of 3, but its *size* is 4.

- Note `'a'` and `"a"` are different. The first is a character constant, the second is a string with 2 elements `'a'` and `'\0'`.

Initialising Strings

- ```
  char w[250];
  w[0] = 'a';
  w[1] = 'b';
  w[2] = 'c';
  w[3] = '\0';
  ```

- `scanf("%s", w);`

  Removes leading spaces, reads a string (terminated by a space or `EOF`). Adds a null character to the end of the string.

- `char w[250] = {'a', 'b', 'c', '\0'};`
  ```

- `char w[250] = "abc";`

- `char *w = "abc";`

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

int main(void)
{
   char w1[100] = "test";
   char *w2 = "test";

   printf("%s -> ", w1);
   w1[0] = toupper(w1[0]);
   printf("%s\n", w1);

   printf("%s -> ", w2);
   /* Seg Faults */
   w2[0] = toupper(w2[0]);
   printf("%s\n", w2);

   return 0;
}
```

Repeated Characters

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

int main(void)
{

   char s[100] =
      "The Quick Brown Fox Leaps " \
      "Over the Lazy Dog";
   short used[26] = {0};
   char c;
   int i = 0;

   while(s[i]){
      c = tolower(s[i]);
      if(islower(c)){
         used[c - 'a'] = 1;
      }
      i++;
   }

   for(i=0; i<26; i++)
      if(!used[i])
         printf("%c has not been used.\n",
                i+'a');

   return 0;

}
```

Output:

```
j has not been used.
m has not been used
```

Pointers & String Processing

```
/* Character processing:
   change a line.         */

#include <stdio.h>

#define   MAXLINE   100

char   *change(char *s);
void   read_in(char s[]);

int main(void)
{
   char   line[MAXLINE];

   printf("\nWhat is your " \
          "favourite line? ");
   read_in(line);
   printf("\n%s\n\n%s\n\n",
    "Here it is after being changed:",
    change(line));
   return 0;
}
```

```
void read_in(char s[])
{
   int    c, i = 0;

   while ((c = getchar()) != EOF &&
            c != '\n')
      s[i++] = c;
   s[i] = '\0';
}

char *change(char *s)
{
   static char   new_string[MAXLINE];
   char          *p = new_string;

   *p++ = '\t';
   for ( ; *s != '\0'; ++s)
      if (*s == 'e')
       *p++ = 'E';
      else if (*s == ' ') {
       *p++ = '\n';
       *p++ = '\t';
      }
      else
       *p++ = *s;
   *p = '\0';
   return new_string;
}
```

Dissection

```
What is your favourite line?
she sells sea shells

Here it is after being changed:

        shE
        sElls
        sEa
        shElls
```

- Every `e` changes to `E`, every space swapped with a newline and tab.
- In `read_in()` we assume less than `MAXLINE` characters are entered. We could use an `assert()` here.
- Note the use of `static` in `change()`.
- `*p++` is equivalent to `*(p++)` and **not** `(*p)++`.

String Handling Functions

In `#include <string.h>` :

```
char *strcat(char *dst, const char *src);
int strcmp(const char *s1, const char *s2);
char *strcpy(char *dst, const char *src);
unsigned strlen(const char *s);
```

- `strcat()` appends a copy of string `src`, including the terminating null character, to the end of string `dst`.

- `strcmp()` compares two strings byte-by-byte, according to the ordering of your machine's character set. The function returns an integer greater than, equal to, or less than 0, if the string pointed to by `s1` is greater than, equal to, or less than the string pointed to by `s2` respectively.

- `strcpy()` copies string `src` to `dst` including the terminating null character, stopping after the null character has been copied.

- `strlen()` returns the number of bytes in `s`, not including the terminating null character.

The Function `strlen()`

One way to write the function `strlen()`

```
unsigned strlen(const char *s)
{

   register int n = 0;

   for(; *s != '\0'; ++s)
      ++n;
   return n;
}
```

Chapter Twelve
Structures

Declaring Structures

- A structure type allows the programmer to aggregate components into a single, named variable.

- Each component has individually named members.

- ```
  struct card {
      int pips;
      char suit;
  };
  ```

- `struct` is a keyword, `card` is the structure tag name, and `pips` and `suit` are members of the structure.
  ```

- A statement of the form :
  `struct card c1, c2;`
  actually creates storage for identifiers

- A member is accessed using the member operator “.”

- ```
  c1.pips = 5;
  c1.suit = 'd';
  c2.pips = 12;
  c2.suit = 's';
  ```

- The member name must be unique within the same structure.
  ```

- We can declare the structure and variables in one statement:

```
struct card {
   int pips;
   char suit;
} c1, c2;
```

- In the above, `card` is optional.

- Arrays of structures are possible, i.e.:

```
struct card pack[52];
```

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

struct student {
   char *surname;
   int id;
   char grade;
};

int failures(struct student j[], int n);

int main(void)
{
   struct student class[] = {
      {"Bloggs",    95112174, 'c'},
      {"Doobeedoo", 96162578, 'b'},
      {"Campbell",  96112103, 'f'},
      {"Johnson",   96185617, 'a'}
   };
   printf("Number of fails : %d\n",
          failures(class, 4));
   return 0;
}

int failures(struct student j[], int n)
{

   int i, cnt = 0;

   for(i=0; i<n; i++)
      cnt += (tolower(j[i].grade) == 'f');
   return cnt;
}
```

Structures & Pointers

- `struct student *p;`
  declares a pointer to a structure.

- Point it at something sensible:
  `p = &c1;`

- Now we access its members using the "`->`" operator.

- `p->id = 96123040;`

- The deference, followed by the member access, could also be written as: `(*p).id = 96123949;`

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

struct student {
   char *surname;
   int id;
   char grade;
};

int failures(struct student *j, int n);

int main(void)
{
   struct student class[] = {
      {"Bloggs",    95112174, 'c'},
      {"Doobeedoo", 96162578, 'b'},
      {"Campbell",  96112103, 'f'},
      {"Johnson",   96185617, 'a'}
   };

   printf("Number of fails : %d\n",
          failures(class, 4));
   return 0;
}

int failures(struct student *j, int n)
{
   int i, cnt = 0;
   for(i=0; i<n; i++){
      cnt += (tolower(j->grade) == 'f');
      j++;
   }
   return cnt;
}
```

```
/* Nested Structures */
#include <stdio.h>

struct date {
   short day;
   short month;
   short year;
};

struct person {
   char name[25];
   struct date date;
};

void print_person_copy(struct person p);
void print_person_point(struct person *p);

int main(void)
{

   struct person k;

   printf("Enter your surname : ");
   scanf("%s", k.name);
   printf("Enter your Birthday: ");
   scanf("%hd/%hd/%hd", &k.date.day,
         &k.date.month, &k.date.year);

   print_person_copy(k);
   print_person_point(&k);

   return 0;
}
```

```
void print_person_copy(struct person p)
{

   printf("Name \"%s\", Born " \
          "%02d/%02d/%d\n",
      p.name, p.date.day,
      p.date.month, p.date.year);

}

void print_person_point(struct person *p)
{

   printf("Name \"%s\", Born " \
          "%02hd/%02hd/%hd\n",
      p->name, p->date.day,
      p->date.month, p->date.year);

}
```

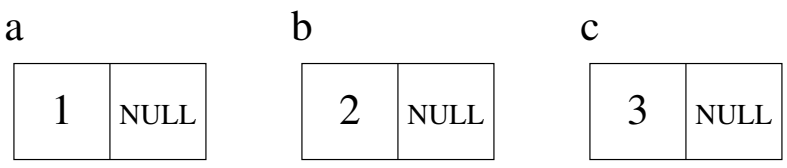
```
Enter your surname : Campbell
Enter your Birthday: 8/12/68
Name "Campbell", Born 08/12/68
Name "Campbell", Born 08/12/68
```

Self-Referential Structures

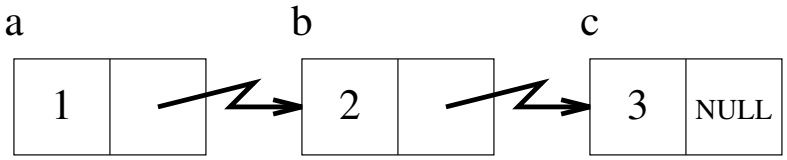
```
struct list {
   int data;
   struct list *next;
};
```

- The pointer variable `next` is called a **link**.

- ```
  struct list a, b, c;
  a.data = 1;
  b.data = 2;
  c.data = 3;
  a.next = b.next = c.next = NULL;
  ```


  ```

- ```
  a.next = &b;
  b.next = &c;
  ```




- ```
  /* has value 2 */
  a.next -> data;
  /* has value 3 */
  b.next -> data;
  /* has value 3 */
  a.next -> next -> data;
  ```

Dynamic List Creation

- We may need to dynamically create objects in a linked list, using `malloc()`.

- `malloc(sizeof(list));` returns a pointer to a new, free block of memory.

- Need to be able to Add/Delete/Count/Insert elements.

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

#define BIGGESTNAME 50

struct llist {
   char word[BIGGESTNAME];
   struct llist *next;
};

int main(void)
{
   struct llist *head, *cp;

   printf("Enter a list of words.\n");
   cp = head = calloc(1, sizeof(struct llist));
   while(scanf("%s", cp->word)==1){
      cp->next = calloc(1, sizeof(struct llist));
      cp = cp->next;
   }

   cp = head;
   printf("The words were :\n");
   while(cp->next != NULL){
      printf("%s\n", cp->word);
      cp = cp->next;
   };
   return 0;
}
```

Chapter Thirteen
File Handling

File Properties

Files have many important properties:

- They have a name.
- Until a file is opened nothing can be done with it.
- After use a file must be closed.
- Files may be read, written or appended.
- Conceptually a file may be thought of as a stream of characters.

Accessing Files

```
#include <stdio.h>

int main(void)
{

   int sum = 0, val;
   FILE *ifp, *ofp;

   /* Read a File */
   ifp = fopen("my_file", "r");
   /* Write to a File */
   ofp = fopen("outfile", "w");

      .
      .
      .

   fclose(ifp);
   fclose(ofp);
}
```

Opening Files

```
FILE *fopen(char *filename, char *modes);
```

The mode for file opening is a *string*, and can take the following options:

“r”	open text file for reading
“w”	open text file for writing
“a”	open text file for appending
“rb”	open binary file for reading
“wb”	open binary file for writing
“ab”	open binary file for appending

If the file cannot be found, or if it is not readable, then the function returns `NULL`.

```
fp = fopen("filedoof", "r");
if(fp == NULL){
   exit(EXIT_FAILURE);
}
```

Getting & Putting Characters

The functions `fgetc()` and `fputc()` are similar to `getchar()` and `putchar()`, but additionally take a file pointer as an argument.

```
int fgetc(FILE *stream);
int fputc(int c, FILE *stream);
```

```
/* Number the lines of a file */
#include <stdio.h>

int main(void)
{

   FILE *fp;
   char fname[128];
   char str[500];
   int i, c, line;

   printf("What file would you like to see ? ");
   scanf("%s", fname);
   if((fp = fopen(fname, "r")) == NULL){
      printf("Cannot open file\n");
      exit(EXIT_FAILURE);
   }
   line = 1;
   do{
      i = 0;
      do{
         c = (str[i++] = fgetc(fp));
      }while(c != EOF && c != '\n');
      str[i] = '\0';
      if(c != EOF){
         printf("%5d %s", line, str);
         line++;
      }
   }while(c != EOF);
   fclose(fp);

   return 0;
}
```

```c
/* Copying Files */
#include <stdio.h>
#include <stdlib.h>

#define MAXFILENAME 128

int main(void)
{

   FILE *ifp, *ofp;
   char inpname[MAXFILENAME];
   char oupname[MAXFILENAME];
   int i;

   printf("Enter two filenames : ");
   if(scanf("%s %s", inpname, oupname) != 2){
      printf("Failed to scan two filenames.");
      exit(EXIT_FAILURE);
   }
   ifp = fopen(inpname, "rb");
   ofp = fopen(oupname, "wb");
   if(ifp == NULL || ofp == NULL){
      printf("Failed to open two filenames.");
      exit(EXIT_FAILURE);
   }
   while((i = fgetc(ifp)) != EOF){
      fputc(i, ofp);
   };

   fclose(ifp);
   fclose(ofp);

   return 0;

}
```

Reading / Writing Data Blocks

The functions `fread()` and `fwrite()` allow blocks of information to be read and written using files.

```
fread(void *ptr, size, num_items, stream);
fwrite(void *ptr, size, num_items, stream);
```

They return an unsigned integer, the number of elements correctly read or written.

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

struct examp {
   int h[100];
   double k;
   char c;
};
typedef struct examp examp;

int main(void)
{

   FILE *fp;
   examp e;
   printf("%d\n", sizeof(double));
   if((fp = fopen("doof", "rb")) == NULL){
      printf("Cannot read file\n");
      exit(EXIT_FAILURE);
   }
   /* Should check return from fread */
   fread(&e, sizeof(struct examp), 1, fp);
   printf("k = %f\n", e.k);
   fclose(fp);
    .
    .
    .
   if((fp = fopen("doof", "wb")) == NULL){
      printf("Cannot write file\n");
      exit(EXIT_FAILURE);
   }
   /* Should check return from fwrite */
   fwrite(&e, sizeof(struct examp), 1, fp);
   fclose(fp);
   return 0;
}
```

Printing and Scanning Files

The functions:

```
int fprintf(FILE *stream, format, other args);
int fscanf(FILE *stream, format, other args);
```

act almost the same as their `printf()` and `scanf()` counterparts, except that input is from file rather than keyboard.

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

int main(void)
{

   FILE *fp;

   if((fp = fopen("file.txt", "w")) == NULL){
      printf("Cannot open file\n");
      exit(EXIT_FAILURE);
   }
   fprintf(fp, "Hello World\n");
   fclose(fp);

   return 0;
}
```

Creates a file called `file.txt` with the string `Hello World` in it.

Stdout and Stderr

On Unix systems, there are three pre-defined streams, `stdin`, `stdout` and `stderr`. Since `stdin` is the standard input stream, `scanf()` is simply as special case of `fscanf()`.

```
scanf("%d", &i);
fscanf(stdin, "%d", &i);
```

Likewise, `printf()` is a special case of `fprintf()`:

```
printf("%s %d", "The number is", i);
fprintf(stdout, "%s %d",
        "The number is", i);
```

For printing error messages use:

```
fprintf(stderr, "Cannot open file %s",
        fname);
exit(EXIT_FAILURE);
```

Even if `stdout` is being re-directed to a file, this message will appear on the screen.

Advanced `scanf()`

It is possible to specify a *scan set* when using `scanf()` to input a string. For instance:

```
scanf("%[abc]s", str);
```

allows you to read in a string containing the characters `a`, `b` or `c`. The string is terminated by any other character, so:

```
Please type a string : abacus
I scanned : abac
```

A circumflex is used to specify only characters *not* in the scan set.

```
scanf("%[^k]s", str);
```

leads to every possible character other than `'k'` being used in the scan set.

```
Please type a string : I Am Crackers
I scanned : I Am Crac
```

The `sprintf()` Function

This is very similar to the function `printf()`, except that the output is stored in a string rather than written to the output. It is defined as:

```
int sprintf(string, control-arg, other args);
```

For example:

```
int i = 7;
float f = 17.041;
char str[100];
sprintf(str, "%d %f", i, f);
printf("%s\n", str);
```

```
Outputs : 7 17.041000
```

This is useful if you need to create a string for passing to another function for further processing.

The `sscanf()` Function

This is similar to the `scanf()` function, except that the input comes from a string rather than from the keyboard. It is defined as:

```
int sscanf(char *s, char *format, other args);
```

For example:

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

#define STRLN 100

int main(void)
{

   char str[STRLN];
   int i;
   fgets(str, STRLN, stdin);
   while(sscanf(str, "%d", &i) != 1){
      printf("Cannot scan string !\n");
      fgets(str, STRLN, stdin);
   }
   printf("The number was %d.", i);

   return 0;
}
```

IO Summary

Keyboard

`scanf(), getchar()`

Screen

`printf(), putchar()`

Output to File

`fprintf(), fputc(), fwrite()`

Input from File

`fscanf(), fgetc(), fread()`

Output to String

`sprintf()`

Input from String

`sscanf()`

Common Errors

- `fopen("myfile", 'r');`
- `double d; scanf("%f", &d);`
- Not using `fclose()`
- Every time `sscanf()` is called, scanning begins at the start of the string, unlike `fscanf()` where a record is kept of how far through the file we are.